

Research Statement

Aleksandar Nanevski

In my research, I explore the principles of programming languages with the goal of improving software reliability, efficiency and maintainability. The cornerstone of my effort is the design and implementation of *type systems and logics* that can specify and reason about precise semantic properties of programs, including safety, security and correctness, while at the same time supporting and building on the most important features of modern programming languages. Such systems are desirable because they can analyze the program statically and discover errors early in program development.

Specifying and reasoning about precise semantic properties is difficult and may require significant amount of work by the programmer. In order to make this process cost-effective and scale it to practical code, it is necessary to split it into manageable chunks and automate it as much as possible. Thus, the main challenge is to derive language designs that: (1) are provably compositional, (2) offer mechanisms for data abstraction and information hiding, so that the amount of reasoning can be kept to a minimum, and (3) enable a degree of automation of the reasoning principles.

In my doctoral and postdoctoral research I have designed type systems for programming with and reasoning about various aspects of stateful programs, while respecting the above three requirements. Handling state is one of the most intriguing and pressing problems in programming languages. Unfortunately, almost all formal systems and techniques for program verification today either target purely functional languages, and break down in the presence of state, or else target imperative languages but do not scale easily to support features like higher-order functions and polymorphism which are necessary for compositionality and abstraction.

The main representatives of systems for pure functional programming come from Dependent Type Theory [3], where types can express arbitrarily complex program properties. The main representatives of systems for reasoning about effectful computations are based around Hoare Logic [1], where program properties are captured by pre- and post-conditions.

The languages that I have developed reconcile these two seemingly disparate approaches in several different ways, and with different applications in mind. They all have in common the use of modal logic to capture in the types the various aspects of program state. The applications that I considered include: reasoning about mutable heaps in the presence of pointer aliasing [11, 10], representation of partial proofs in unification and logic programming [13, 14], type systems for staged computation and metaprogramming [12, 4], and tracking of computational effects related to a store, dynamic binding and control flow [5, 7, 6].

In my future work, I intend to improve the usability of the technology for the listed applications, investigate further its theoretical underpinnings, expand into other application domains like concurrency, security and protocol verification, and test the technology by applying it in the development and verification of practical software systems.

Reasoning and programming with state

My most recent work is related to the development of Hoare Type Theory (HTT) for reasoning about dynamically allocated data and pointer aliasing [11, 10] in the style of Hoare Logic and more recently, Separation Logic [15, 16]. The main idea of HTT is to turn specifications in the form of Hoare triples into types. The Hoare triple type $\{P\}x:A\{Q\}$ classifies programs that can execute in a state satisfying the assertion P and either diverge, or terminate with a value x of type A in a state satisfying the assertion Q .

Using specifications as types has numerous benefits, because it leads to an integrated development of a program and its specification. For example, HTT can nest the specifications, combine them with other types, and abstract over them, thus improving upon the information hiding mechanisms of the original Hoare Logic. Second, HTT safely combines type theoretic features such as higher-order functions, polymorphism and modules, with low-level features such as explicit memory management and pointer arith-

metic. Third, HTT supports modular specifications in the style of Separation Logic, whereby the pre- and postconditions of a program need only describe the state that the program actually uses; the unspecified state is automatically assumed invariant.

HTT can also be seen as a provably sound and compositional formalization of the core features of systems like ESC/Java, Spec#, JML, and Cyclone which support extended static checking of programs.

As with any sufficiently rich specification systems, checking that HTT programs respect their types is generally undecidable. However, it is possible to split the checking into two independent phases. The first performs a combination of basic type-checking and verification-condition generation, both of which are carefully designed to be decidable. The second phase must then show the validity of the generated verification conditions. These conditions can either be ignored, fed to an automated theorem prover, or discharged by hand. This makes it possible to provide various levels of correctness assurance within the same system. For example, because of the limitations of the proving technology, using an automated prover is unlikely to discharge all of the conditions, but may be strong enough to find a number of programming errors. Alternatively, proving by hand or using an interactive theorem prover, can lead to a formal certificate that the program is error-free.

Modal logic and types. The theoretical underpinnings of HTT lie in my doctoral research on modal logic and modal types [6]. Modal logic is designed for reasoning across various *abstract* worlds. It features two type constructors: \Box (box) and \Diamond (diamond). The constructor \Box is a universal quantifier: $\Box A$ is true in the current world iff A is *necessary*, i.e. true in *all* the worlds. The constructor \Diamond is an existential quantifier: $\Diamond A$ is true in the current world iff A is *possible*, i.e. true in *some* world.

For computational purposes, it is useful to think of worlds as standing for program state. Then, programs with a precondition P can be ascribed *bounded universal types* $\Box_P A$, because they can execute in *all* states satisfying the assertion P . Programs with a postcondition Q can be ascribed a *bounded existential type* $\Diamond_Q A$, because they produce *some* state satisfying the assertion Q . The Hoare type in HTT is essentially a composition of the modalities \Box_P and \Diamond_Q . However, it may sometimes be beneficial to consider the two modalities in isolation, as illustrated below.

Control flow effects. A program that can raise an exception, requires a precondition that a handler for the exception is installed. If the precondition is satisfied, the program will produce a value or diverge, but will not get stuck. Thus, specifying an exceptional computation requires a precondition, but not necessarily a postcondition, as no program state is changed [7]. This is an example where we can use the modality \Box_P without the combination with \Diamond_Q . Similar observation can be made about other control effects like catch/throw and composable continuations [6].

State reads and writes. Let X be a name of an allocated (but perhaps uninitialized) memory location, and consider an expression that reads from X before returning a value of type A . Such an expression may be ascribed a bounded universal type $\Box_X A$, because it can compute a value in any environment in which X is initialized. Dually, an expression that writes into X , before returning a value of type A may be ascribed a bounded existential type $\Diamond_X A$. Such an expression is a witness that there exists an environment (the one obtained by writing into X) in which a value of type A can be computed.

In this sense, modal types differentiate between location reads and location writes, but also structure the interaction between the two [5]. This distinction may be very useful. For example, it is usually possible and desirable to execute a number of location reads out of order, and even in parallel, while location writes must typically be serialized.

As it turns out, the fragment of this modal calculus, containing only the \Box operator provides a logical explanation for *dynamic binding*.

Staged computation, unification and partial proofs. In staged computation and run-time code generation, programs are composed at run time out of smaller programs. A good language for this application must provide an operation for *capture-incurring* substitution of expressions with free variables into a larger context. This is yet another instance of programs with preconditions, but no postconditions. An expression of type A with a free variable X , may be ascribed a universal type $\Box_X A$, because it may be substituted into *any* context capable of capturing X [4, 12]. A similar operation is required of algorithms for pattern

unification in logic programming. Here modal types offer a novel view of unification variables, leading to a logical explanation and generalization of many previously considered optimizations, like lowering, raising and grafting [14, 13].

Future research

The long term goal of my research is to develop the programming language technology based on expressive logical formalisms and type theory to the point where such systems could be used to automatically, or with minimal amount of programmer interaction, certify the correctness of real-world programs. Towards achieving this goal, I plan to focus on the following several directions.

Improving the usability of systems with expressive specifications. Working with expressive specifications is cumbersome. In order to facilitate the adoption of this kind of language technology, I plan to consider two different strategies.

The first strategy is to identify problems where the technology can be completely automated. An example is my work on contextual modal type theory for staged computation and unification [13, 12]. In this application, the preconditions are described by finite sets of variables, rather than by arbitrary assertions, and this leads to decidable typechecking.

The second strategy is to design the systems so that they can be useful in finding errors even if the programmer does not care to invest the time and effort to obtain the full correctness. An example is the design of HTT [11, 10], which can provide various levels of correctness assurance, as I described before.

A usable language must also offer effective type and invariant inference, and a useful and precise reporting of programming errors. For example, my coauthors and I report some initial results on inferring loop invariants in Separation Logic in [2]. This kind of questions have been considered previously for systems for extended static checking, and involved a host of formal methods including theorem proving, abstract interpretation, predicate abstraction and model checking. The previous work only investigated first-order languages, but I plan to focus on the higher-order setting.

Most type theoretic languages and theorem proving systems (HTT included) support infinite precision natural numbers, or other forms of algebraic numerical structures. While infinite precision arithmetic may be very useful for certain applications [9], most software will require bounded integer and floating-point arithmetic. I would thus like to investigate axiomatizations and automation of reasoning principles for this kind of numerical domains, possibly employing techniques from formal methods and SAT solving.

I also plan to investigate new language constructs for data abstraction and information hiding, which would reduce the work involved in program verification. For example, HTT currently supports explicit memory management and pointer arithmetic. However, if one is only interested in pointers that cannot be deallocated and do not support arithmetic (e.g., like in the mainstream functional languages), then the size of verification conditions can be significantly reduced. Similar observations can be made about objects and abstract data structures which hide their local state.

Studying the theoretical properties. Understanding the logical properties of systems with expressive specifications will help the deployment in new settings. Some questions that I would like to study involve the axiomatization and addition of other stateful features, like exceptions, I/O, as well as reasoning about the file system, and other OS resources. It is an open question in (monadic) functional programming how various notions of effects can be combined and tracked through types.

I would also like to study and develop methodology for reasoning about equivalence of stateful programs. A principal example is the technique of logical relations, whose definition essentially relies on the type system in question, and which has been notoriously hard to extend to stateful programs. I believe that considering expressive type system may lead to advances in this area.

A related problem is proving the parametricity of local store [8]; that is, if a fragment of store is accessible only within a procedure, then changing the store layout while preserving the procedure specification should not influence the outcome of the program. Similar considerations appear in the study of languages and logics for reasoning about the flow of secure information through programs.

Expanding into other application domains. HTT presents a methodology for converting a Hoare-like logic into a programming language with expressive types. It may be possible to take a Hoare logic for some domain specific safety-critical application, (e.g, reasoning about concurrency, security or resource bounds) and employ the HTT methodology to obtain a language for constructing provably correct and compositional programs. I am also interested to explore how other logics can be turned into type systems, with similar goals in mind. Some examples include logics for authorization and protocol verification.

References

- [1] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [2] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE'06*, pages 47–60, Charleston, South Carolina, January 2006.
- [3] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.
- [4] A. Nanevski. Meta-programming with names and necessity. In *International Conference on Functional Programming, ICFP'02*, pages 206–217, Pittsburgh, Pennsylvania, 2002.
- [5] A. Nanevski. From dynamic binding to state via modal possibility. In *International Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages 207–218, Uppsala, Sweden, 2003.
- [6] A. Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Computer Science Department, Carnegie Mellon University, June 2004. Available as Technical Report CMU-CS-07-9254872.
- [7] A. Nanevski. A modal calculus for exception handling. In *Intuitionistic Modal Logic and Applications Workshop*, Chicago, June 2005.
- [8] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. Technical Report TR-14-06, Harvard University, July 2006.
- [9] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. *Higher-Order and Symbolic Computation*, 16(4):379–400, December 2003.
- [10] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, December 2005.
- [11] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming, ICFP'06*, pages 62–73, Portland, Oregon, September 2006.
- [12] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, November 2005.
- [13] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. Revised version to appear in *ACM Transactions on Computational Logic*, September 2005.
- [14] A. Nanevski, B. Pientka, and F. Pfenning. A modal foundation for meta-variables. In F. Honsell, M. Miculan, and A. Momigliano, editors, *Workshop on Mechanized Reasoning about Languages with Variable Binding, MERLIN'03*, pages 159–170, Uppsala, Sweden, 2003.
- [15] P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
- [16] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science, LICS'02*, pages 55–74, 2002.