Research Statement

Aleksandar Nanevski

In my research, I explore how expressive logical formalisms can serve as foundations for novel programming languages. Languages based on logic facilitate easier discovery of programming errors and development of more efficient, robust and maintainable software. I am also interested in designing methods and tools for transformation, analysis and verification of programs in such languages.

My recent focus has been on investigating how programs interact with their execution environments. This question is becoming increasingly important, as complex and diverse execution environments are becoming more common. Indeed, programs today are often required to be mobile, run in parallel, use distributed data owned by different authorities, and accommodate changing run-time conditions. As the environments in which programs execute are becoming more complex, so is programming.

In my dissertation [4], I explored how programming languages can help tame this complexity. The main idea is to use various modal logics that can capture statically the properties of execution environments. I applied my approach to several different areas: metaprogramming, logic programming, and the tracking of computational effects. In each of these areas, my approach produced simpler and more general solutions to some long-standing problems.

Motivation

When approaching a difficult programming problem, a language-enforced programming discipline is crucial, because it restrains the programmer from writing unreadable or incorrect code. A natural way to enforce this discipline is through the type mechanism of functional languages. Types express assumptions and guarantees required of expressions, and usually correspond to propositions in some logic. The compiler can mechanically check if the expression matches its specified type, thereby aiding the debugging process.

Type systems of most languages today ensure that functions are invoked with matching arguments but, unfortunately, ignore how programs interact with execution environments. In order to manage the increased complexity of programming, a language-enforced typing discipline that takes environments into account is a critical component. Indeed, if types could capture important aspects of execution environments, then the type system could also ensure that expressions are always executed in matching environments.

What does it mean for an expression and an environment to match? As an illustration of the concept, consider the following example. Assume that an execution environment consists of a number of allocated, but not necessarily initialized memory locations. An expression interacts with this environment by reading or writing into the locations. One possible definition of matching may, for example, insist that each expression reading from a number of locations is always executed in a state of memory where these locations are actually initialized.

The definition of matching may be instantiated in different ways for different applications. However, as I will now describe, there is a thread unifying all of them – namely, the use of modal logic and modal types to specify how expressions and environments may interact.
Modal logic and modal types

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, we can think of worlds as standing for execution environments; then, expressions that depend on the environment will be given universally quantified types, and expressions that change the environment will be given existentially quantified types, as illustrated below.

State reads and writes \hspace{-0.3cm} 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 $\Diamond_X A$. Indeed, 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 [3]. 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. The modal type system ensures that only initialized locations are ever read, and that there are no race conditions between reads and writes. These guarantees may enable more aggressive optimizations.

As it turns out, the fragment of this modal calculus, containing only the $\Box$ operator is actually a calculus of dynamic binding [3]. Discovering a logical foundation behind dynamic binding was a long-standing problem in functional programming.

Control flow effects \hspace{-0.3cm} Let $X$ be a name of an exception and consider an expression that may raise $X$ before returning a value of type $A$. This expression can compute a value in any environment capable of handling the exception $X$, and thus may be ascribed a bounded universal type $\Box_X A$ [1]. A benefit of the modal approach is that the exceptional computations need not be serialized. This is not the case in the alternative, monad-based, treatment of exceptions.

Automatic program generation \hspace{-0.3cm} In run-time code generation and metaprogramming, programs are composed at run time out of smaller programs. A good language for this application must provide an operation for capture-incuring substitution of expressions with free variables into a larger context. This is yet another instance of the interaction with environments. 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 $X$ [2, 6]. 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 [8, 7].

Applications

I had an opportunity to explore applications of run-time code generation and metaprogramming within the framework of the Parallel and Scientific Computation (PSciCo) project at Carnegie Mellon. It is in fact this research that led to my work on modal type systems. As a member of the
PSciCo project, I designed and implemented a compiler for automatic generation of staged geometric predicates [5]. In geometric algorithms it is frequently necessary to determine the sign of a polynomial function at some point, in order to decide whether the point lies in/out/on a line, circle, plane, sphere, etc. Floating-point arithmetic alone does not suffice for this computation, as the rounding errors can perturb the sign. Exact arithmetic guarantees a correct result, but it is slow. The usual compromise is to carry out the computation in floating-point first, estimate the rounding error, and employ exact arithmetic only if the estimated error is too big to reliably determine the sign. A particularly efficient instance of this idea is to introduce several computational phases, each reusing the results from the previous one, and refining its accuracy. Unfortunately, these phases cannot be described in a uniform way, as they do not consist of always re-running the same program with higher and higher precision. Therefore, developing geometric predicates in this style by hand is tremendously difficult, time consuming and error-prone. Our compiler generates all of the phases automatically, with efficiency of the generated predicates reasonably close to hand-tuned versions (in the rare cases when hand-tuned versions existed).

Another area that I have recently started to explore, is the verification of C programs by means of model checking techniques like predicate abstraction. I have specifically focused on programs with memory allocation, deallocation, mutation and pointer arithmetic. Such programs are notoriously difficult to verify because the possibility of pointer aliasing causes an explosion in the number of conditions that must be checked. However, a significant breakthrough has recently been made by Pym, Ishtiaq, O'Hearn and Reynolds with the discovery of separation logic for resource management. In this logic, it is possible to reason about memory heaps in a very modular way and consequently, invariants of programs with pointers can be stated very concisely. It is an interesting open question if separation logic can be effectively used in model checking tools to automatically verify programs with pointers. This is part of my ongoing research as a postdoctoral fellow at Carnegie Mellon.

Future research

I intend to continue my study of the logical foundations of programming languages and apply it to practical algorithmic and systems issues. In the more immediate future, I plan to focus on the following questions concerning modal logic and modal types.

Modal types for distributed computation, security, resource bounds and ownership By changing what the indices on modal types mean, it may be possible to obtain languages for many diverse applications. For example, the type \( \Diamond_X A \) may stand for: (a) expressions executable on all networked computers that provide the resource \( X \), or are owned by the authority \( X \); (b) computations encrypted by the key \( X \); (c) a parallel process that reads from the channel \( X \).

Modal types in practice I would like to integrate my research on bounded modal types into a realistic language, such as ML, Haskell or Java. The goal of the integration is to explore the practical benefits of the richer type structure as it relates to run-time code generation, metaprogramming, combination of various computational effects, and other concepts amenable to modal treatment. Richer types will facilitate more aggressive optimizations and identify more programming errors at compile time. The practicality of this integration will depend on the possibilities for efficient type inference, and on the additional language features like polymorphism, type refinements, recursive types, etc.

Metaprogramming, automatic generation and inspection of programs The need for automatic generation of code arises naturally whenever a general programming solution needs to be special-
ized to a particular set of inputs. For example, in scientific computing efficiency can benefit from the specialization, but the specialization may either be hard to generate by hand [5], or must be generated at run time. There are many outstanding questions in this area. For example, should we generate specialized code quickly, or should we generate fast code? A modal programming language for code generation and inspection is a perfect framework for addressing these issues [2, 6].

Other uses of modal types arise whenever we need to algorithmically manipulate syntactic formal objects (such as programs or proofs) that are written in other formal languages. The challenge here is to ensure that various formalisms do not accidentally interfere. When all the involved languages are typed, this amounts to ensuring that well-typed user programs always produce well-typed formal objects. Examples of this kind that may benefit from modal logic include operating system scripting languages, languages for proof scripting in automated theorem proving, logical frameworks [8, 7], and systems for symbolic and algebraic computation like Mathematica or Maple.

**Reasoning about modal programs** Modal types offer a rich structure capable of capturing computational concepts from very diverse application domains in a rather uniform way. The uniformity makes it plausible that common formal methods for representing, reasoning about, and verifying modal programs could be identified and developed. A dependent modal type theory [8, 7] is a likely framework for such an investigation.

**References**


