Research Statement

Brigitte Pientka

My principal research interest lies in developing a theoretical and practical foundation for building and reasoning about reliable software. To achieve this goal, I combine theoretical research in programming languages and verification with system building and real-world experiments. In particular, I apply techniques from logic, type theory, and automated deduction to find rigorous solutions to problems exposed in practice. The main focus of my work has been the logical framework Twelf, which provides an experimental platform to specify, implement, and execute formal systems. Such formal systems, described by axioms and inference rules, include operational semantics of programming languages, type systems, or different logics. To concisely model many features prevalent in these formal systems the Twelf system supports a typed higher-order logic programming language. The high expressive power of this programming language together with its declarative nature makes Twelf an ideal foundation for prototyping and building reliable software.

Recently, Twelf has been applied to mobile code security in large-scale projects at Princeton and Carnegie Mellon. To provide guarantees about the behavior of mobile code, programs are equipped with a certificate (proof) that asserts certain safety properties. These safety properties are represented as higher-order logic programs. Twelf's logic programming interpreter then executes the specification and generates a certificate (proof) that a given program fulfills a specified safety policy. Experience with these applications has increasingly demonstrated that the practicality of the system critically depends on two issues: first, its ability to execute straightforward, simple specifications and second, to perform well and consistently even in large-scale examples.

A central goal in my thesis has been to develop techniques to overcome existing performance barriers and extend its expressive power. Through my work, performance is improved by several orders of magnitude and sustained even in large-scale examples. This is a significant step toward exploring the full potential of logical frameworks in real-world applications. This approach, I believe, leads to a deeper understanding of the overall design and implementation of reliable, secure, and efficient software systems.

In the following I will briefly describe some of my contributions in the areas of logic programming, efficient data-structures and algorithms, and verification. Finally, I will give an outlook on applications and future work.

Higher-order logic programming

The impact of logical frameworks in practice has been limited for two main reasons: First, their performance may be severely hampered by redundant computation, leading to long response times and slow development of formal specifications. Second, many straightforward specifications of formal systems, for example recognizers and parsers for grammars, have not been executable, thus requiring more complex and often less efficient implementations. A central contribution of my thesis is a novel execution model for logical frameworks based on selective memoization, which extends the expressive power and eliminates some critical performance barriers. Memoizing common subproofs and re-using their results later allows the elimination of redundant and infinite computation. This has three advantages. First, proof search is faster thereby substantially reducing the response time to the programmer. Second, the proofs themselves are more compact and smaller. This is

especially important in applications to secure mobile code where a proof is attached to a program, as smaller proofs take up less time to check and transmit to another host. Third, substantially more specifications, for example recognizers and parser for grammars, are now executable under the new paradigm thereby extending the power of the existing system. In [1], I present a proof-theoretic characterization of this execution model and prove soundness of the resulting interpreter. Based on this theoretical work, I have implemented a prototype, which is now part of the main Twelf distribution and conducted several experiments with parser and recognizers for grammars, refinement type system, and rewriting systems. Preliminary experimental results have been presented in [2]. This work, I believe, is an important step toward a practical programming environment for developing and prototyping formal systems, thereby facilitating the design of reliable software systems in general.

Efficient data-structures and algorithms

Efficient data-structures and implementation techniques play a crucial role in utilizing the full potential of a reasoning environment in real-world applications. Although this need has been widely recognized for first-order languages, efficient algorithms for higher-order languages are still a central open problem. My contributions in this area are two-fold.

Higher-order term indexing. Proof search strategies, such as memoization, can only be practical, if we can access the memo-table efficiently. Otherwise, the rate of drawing new conclusions may degrade sharply both with time and with an increase of the size of the memo-table. Term indexing aims at overcoming program degradation by sharing common structure and factoring common operations. Higher-order term indexing has been a central open problem, limiting the application and the potential impact of higher-order reasoning systems. I have designed and implemented higher-order term indexing techniques. They improve performance by up to a factor of 10, illustrating the importance of indexing. (paper is in progress)

Optimizing unification. Unification lies at the heart of logic programming, theorem proving, and rewriting systems. Thus, its performance affects in a crucial way the global efficiency of each of these applications. Higher-order unification is in general undecidable, but decidable fragments, such as higher-order patterns unification, exist. Unfortunately, the complexity of this algorithm is still at best linear, which is impractical for any useful programming language or practical framework. I have designed an assignment algorithm, which takes constant time, by factoring out unnecessary occurs checks. Experiments show that we get a speed-up by to a factor $2 - 5$ making the execution of some examples feasible. This is a significant step toward efficient implementation of higher-order reasoning systems in general. (paper is in progress)

Verification

Software verification is a central aspect to my research goal of developing reliable and correct software. It provides the highest level of assurance that the program is correct and secure. However, it also can be time consuming, expensive, require manual interaction, and may not even be feasible in distributed environments. On the other hand verifying invariants about the run-time behavior of programs can be done quickly, cheaply and completely automatically. This approach guarantees partial correctness and it allows us to catch flaws early on in the design process, thereby increasing

the confidence in the correctness of the systems.

My research interest spans the whole spectrum of verification described above and my earliest research experiences at the University of Edinburgh and the Technical University of Darmstadt have been in this area [3, 4, 5]. During my research visit at Cornell University, I collaborated with Christoph Kreitz on verifying program specifications and developed techniques for induction theorem proving in the NuPRL system (see [6, 7, 8, 9, 10]).

At Carnegie Mellon I worked on describing and checking invariants about the run-time behavior of programs. An example of such an invariant is that the execution of a program terminates, or even stronger the execution terminates within a given bound. In addition to termination, we might also want to give certain guarantees about the result of the program execution, for example that the result is strictly smaller than the input or at least that it is non-size increasing (reduction property). I developed a proof-theoretic characterization for reasoning about structural properties and proving termination and reduction for higher-order logic programs (see [11, 12, 13]). The termination and reduction checker extends and generalizes previously used methods for termination checking and is implemented as part of the Twelf system.

Application and future work

I believe it is vital to combine theoretical research with practical applications. Such synergy often provides new insights and allows the validation of theoretical results. As mentioned earlier, one of the applications of my research lies in mobile code security. I have had the opportunity to evaluate some of the above described research using an implementation of proof-carrying code, which Andrew Appel's research group at Princeton is developing using Twelf. Experiments with their benchmarks show substantial performance improvements and significantly reduce development time.

Another central question motivated by proof-carrying code is concerned with proof checking. In practical applications the size of proof can be large. Hence a crucial problem is how to reduce the time to transmit and check a proof. Based on Necula's work in proof-carrying code, I have started collaborating with Karl Crary and graduate student Susmit Sakar to encode proofs as bit-strings which encode the non-deterministic choices a higher-order logic programming interpreter makes. On the producer side, a higher-order logic programming interpreter generates this bit-string. On the consumer side, the logic programming interpreter is guided by the bit-string to check that the program is safe. In the future, I intend to continue to explore applications such as proof-carrying code and investigate new ones in areas such as authentication and security.

My long-term research plan is to develop the logical framework Twelf toward a practical system that supports debugging and prototyping. In a practical system short response time to the programmer and precise error messages are critical. In the near future I plan on continuing my work in this area. Here are several concrete examples I plan on working.

Compilation and optimizations in logic programming. I have already made contributions in this area. In the future, I am particularly interested in exploiting invariants about programs such as termination, determinism, and mode properties during compilation and execution.

Theorem proving and logical frameworks. I have had the opportunity to be involved in the development of different theorem proving systems at Technical University of Darmstadt, University of Edinburgh, Cornell University and Carnegie Mellon. Based on this experience, I plan to apply automated theorem proving techniques to proof search in Twelf.

Linear logic programming. Linear logic programming has been proposed as an extension of higher-order logic programming to model imperative state changes in a declarative (logical) way. It has been successfully applied to represent Petri-nets, planning problems, and CCS transition systems. However, performance may be severely hampered limiting the impact of linear logic programming. I believe the memoization techniques can be extended to the linear case to tackle some of the performance problems, but this requires some new considerations.

References

[1] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), pages 271 –286. Springer-Verlag, 2002.

[2] Brigitte Pientka. Memoization-based proof search in LF: an experimental evaluation of a prototype. In *Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02), Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science (ENTCS), to appear, 2002.

[3] Stefan Gerberding and Brigitte Pientka. Structured incremental proof planning. In *Proceedings of the 21th German Annual Conference on Artificial Intelligence, Freiburg, Germany*. Springer-Verlag, 1997.

[4] Brigitte Pientka. Structuring and optomizing incremental proof planning. Master's thesis, Technical University Darmstadt, 1997.

[5] Brigitte Pientka. A heuristic for case analysis. Technical Paper 37, Departement of Artificial Intelligence, University of Edinburgh, Scotland, 1995.

[6] Christoph Kreitz and Brigitte Pientka. Connection-driven inductive theorem proving. *Studia Logica*, 69(2):293–326, 2001.

[7] Christoph Kreitz, Jens Otten, Stephan Schmitt, and Brigitte Pientka. Matrix-based constructive theorem proving. In Steffen Hölldobler, editor, *Intellectics and Computational Logic. Papers in honor of Wolfgang Bibel*, number 19 in Applied Logic Series, pages 189–205. Kluwer, 2000.

[8] Christoph Kreitz and Brigitte Pientka. Matrix-based inductive theorem proving. In R. Dyckhoff, editor, *9th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, Lecture Notes in Artificial Intelligence (LNAI) 1847, pages 294–308. Springer Verlag, 2000.

[9] Brigitte Pientka and Christoph Kreitz. Automating inductive specification proofs. *Fundamenta Informatica*, 39(1–2):189–209, 1999.

[10] Brigitte Pientka and Christoph Kreitz. Instantiation of existentially quantified variables in inductive specification proofs. In J. Plaza and J. Calmet, editors, *4th International Conference on Artificial Intelligence and Symbolic Computation*, Lecture Notes in Artificial Intelligence (LNAI) 1476, pages 247–258. Springer-Verlag, 1998.

[11] Brigitte Pientka. Termination and reduction checking for higher-order logic programs. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Prooceedings of the first International Joint Conference on Automated Reasoning, Siena, Italy*, Lecture Notes in Artificial Intelligence (LNAI) 2083, pages 401–415. Springer-Verlag, 2001.

[12] Brigitte Pientka. Termination and reduction checking in the logical framework. Technical report CMU-CS-01-115, Carnegie Mellon University, 2001.

[13] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. In C. Schürmann, editor, *Workshop on Automation of Proofs by Mathematical Induction, Pittsburgh, PA*, 2000.