Research Statement

Frank Fu

The goal of my research is to develop effective methods to help programmers writing correct programs. One way to approach this goal is through adopting type systems and functional programming languages. Functional programs have the advantage of being concise and enabling equational reasoning. Type systems provide a form of light-weight language-based verification, i.e., by writing down a compilable program, the programmer knows that the program will not run into unknown state. My research focus is on **developing type systems to ensure various properties of functional programs**.

Type system provides ideal abstraction for both theoretical study and practical implementation of functional programming languages. It has been a challenging task to balance functional programming, theorem proving and automation in a type system. To address this challenge, I investigated different type systems for certification, theorem proving and functional programming. Some of these investigations have resulted in novel methods to some old problems.

Types for describing rule-based computation

First-order systems such as term rewriting system [1] and logic programming by Horn formulas [10] are two common frameworks for describing rule-based computation. Term rewriting works by matching the rewrite rule against a given term and reducing it to another term. Logic programming is similar, with Horn formulas serving as rules, which are unified against a given goal to resolve it to several subgoals. The computation process given by term rewriting or logic programming can be nonterminating, in which case the computation is often considered as meaningless and a situation to avoid.

I propose to represent first-order rules as types and computation as terms in a suitable type system. The nonterminating computation arises from logic programming or term rewriting can be understood by the fixpoint typing rule. Thus the nonterminating process is meaningful and subject to certification. This idea is realized in the following papers.

- **Functional certification of nonterminating rewriting** [6]. Rewrite rule and first-order term are represented as types, the nontermination of a first-order term can be certified by a proof term that contains fixpoint. The certification of nontermination is reduced to the second-order type checking, hence resulting a novel example of the application of second-order type checking.
- **A Type theoretic approach to logic programming** [5]. The standard SLD-resolution for logic programming can be modeled by a first-order simply typed system, where Horn formulas corresponds to types, the resolution process corresponds to functional composition. Based on the type-theoretic framework, different forms of resolution based on different uses of matching and unification are identified and discussed.
- **Proof relevant corecursive resolution** [7]. The first-order simply typed system is extended with the fixpoint typing rule, the resulting type system can provide certification for some nonterminating SLD-resolution by matching. In the context of Haskell type class, the certification corresponds to dictionary construction, we show how to apply this approach to solve the nontermation problem in Haskell's type class instance resolution.

I have implemented two proof-of-concept prototypes[1],[2] to demonstrate the use of type theoretic approach

[1] `https://github.com/fermat/corecursive-type-class`
[2] `https://github.com/fermat/fcr`

for describing first-order computation. I plan to continue my efforts to further advance the application of using type system to provide accurate and executable certification for various kinds of computation.

Types for reasoning about functional programs

According to the Curry-Howard correspondence [9], types correspond to formulas, terms correspond to proofs. It is not uncommon to use a type system for both functional programming and logical reasoning. For logical reasoning, we need to show that the type system is consistent as a logical system. For functional programming, we want to be able to easily write down programs that run with the expected efficiency. However, to achieve these two goals within a single type system while having a satisfying metatheory is very challenging. Currently there are two approaches to this problem: (1). Use one type system for both reasoning and programming. (2) Use two type systems, one for reasoning and the other for programming, where the logical fragment can reason about the programming fragment. The benefit of approach (1) is that there is only one type system to implement. The drawback of approach (1) is that as a reasoning system, showing its consistency is challenging; as a programming language, it is not flexible to write general recursive functional programs. The benefit of approach (2) is that it is standard to prove logical consistency and it is easy to write general recursive functional programs. The drawback of approach (2) is that there are two separated type systems to implement.

I have experience on working with both approach (1) and (2).

- **Self type for dependently typed lambda encoding** [8]. A novel type construct called *self type* is introduced as one of the extensions to the standard Calculus of Construction [3]. A type system called **S** that allows programming with terminating recursion by lambda encodings is defined and it is shown to be both type preserving and consistent. System **S** can be viewed as a realization of approach (1) with reasonable compromise on the programming style and the logical expressivity. This work has generated some interest in investigating the efficiency of different lambda encoding schemes in a total type theory [11].

- **Lambda encoding with comprehension** [4]. Working with a version of higher-order logic with explicit comprehension principle and the equality theory of lambda calculus, I implement a prototype system called Gottlob[3] that allows the usual polymorphic typed functional programming (program can be nonterminating) and the sound reasoning on these programs (the logical system is consistent). Gottlob can be viewed as a realization of approach (2), with a reasonable compromise of having essentially two independent type systems.

Research Agenda

I am always interested in developing new cost-effective verification methods that can be integrated into current type system and functional programming paradigm. In particular, I plan to explore the following research directions.

- **Resolution-based type checking** Resolution-based type checking algorithm shows promise to scale to second-order types [6], the type annotation requirement for resolution-based algorithm is well-understood. The application of second-order type checking is less explored, I am seeking opportunities to use it to reduce the duplication of programs. I am working on devising a type system to allow resolution-based second-order type checking, as well as to accommandate additional features such as higher-rank types, type class and GADTs [2].

- **Language-based verification for the compiler** After the Gottlob project, I realize that the verification of the program is based on the implemented interpreter (evaluator). But in order to be useful, the verified program will be compiled to assembly code, then the property of the program will not

[3] `https://github.com/fermat/gottlob`

be guaranteed as the compiler may contain bug. As a result, all the verification effort is lost. One can certainly prove the compiler is correct[4], but this is not considered a light-weight approach. I am planning to explore methods to target the verification ability of the language to the compiler instead of the interpreter.

References

[1] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[2] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[3] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[4] Peng Fu. *Lambda Encodings in Type Theory*. PhD thesis, University of Iowa, 2014.

[5] Peng Fu and Ekaterina Komendantskaya. A type-theoretic approach to resolution. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 91–106. Springer, 2015.

[6] Peng Fu and Ekaterina Komendantskaya. Functional certification of nonterminating rewriting. *In submission*, 2016.

[7] Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. Proof relevant corecursive resolution. In *International Symposium on Functional and Logic Programming*, pages 126–143. Springer, 2016.

[8] Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In *International Conference on Rewriting Techniques and Applications*, pages 224–239. Springer, 2014.

[9] W. A. Howard. The formulæ-as-types notion of construction. In Philippe De Groote, editor, *The Curry-Howard Isomorphism*. Academia, 1995.

[10] U. Nilsson and J. Małuszyński. *Logic, programming and Prolog*. Wiley Chichester, 1990.

[11] Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming*, 26, 2016.

[4] E.g. `http://compcert.inria.fr/`