Proving Program Correctness in Compilation Time

Theses of PhD Dissertation

Gergely Dévai
MSc in Computer Science
deva@elte.hu

2017

Supervisor: Zoltán Csörnyei, PhD, associate professor
Eötvös Loránd Univeristy of Sciences, Faculty of Informatics,
Pázmány Péter sétány 1/C, 1117 Budapest, Hungary

PhD School of ELTE Fac. Inf.
PhD program: Basics and methodology of informatics
Leader of the PhD School: Erzsébet Csuhaj-Varjú, PhD
Leader of the program: János Demetrovics, PhD, academician

Introduction

Developing error-free programs has been a goal from the beginnings of software development. The most radical solution to provide correctness is to prove it formally. Despite the long history of formal methods these techniques are used in a small segment of the software industry only. The reason for this is that developing formal proofs requires much more effort than creating the software itself, even if one uses machine support.

On the other hand software safety gets more and more important as programs play key roles in more and more critical fields of our lives: they control our transport vehicles, lifesaving equipments, financial transactions etc. In addition there is another reason for creating error-free software: Maintaining and debugging software products cost far more then developing the product itself, and this cost can be reduced by the application of formal methods.

Is there any hope that formal methods in software development get much more widely used? Let us ask another question: Was it reasonable, in the epoch of machine code on punch cards, to believe in an enormous amount of software, supporting all aspects of our lives, created within a few decades by only a relatively small number of software developers? As of now, we already know that the advances of software technology made this possible by providing methods like compilation, object orientation, model based development and functional programming etc. It is possible that the software technology of the next couple of decades will make formal software development as cost efficient as the development of proofless programs nowadays.

Goals and Research Directions

The goal of my PhD research was to apply techniques resulting from the evolution of programming languages to prove the formal correctness of programs. The concrete goal was to elaborate a programming language that can be used to express formal specification of a program and the proof of correctness of a program fulfilling the specification. The tasks of the compiler of the language are to verify the proof and, given that it is sound, extract the program from the proof automatically.

To reach this goal it was necessary to generalise the existing formal programming models and to elaborate the required language features.

The research results has shown that the tool-set of metaprogramming is useful for the efficient development of proofs. In case of embedded languages the so-called host language (the programming language that is used to create the embedding) can be used for metaprogramming purposes on top of the embedded language. For this reason I have created an embedded version of the language.

Functional programming languages are widely used as host languages. On the other hand, my research made it clear that the usual pattern matching features of mainstream functional languages are not ideal for the analysis of embedded language entities. For this reason I have proposed an extension to pattern matching. This result is universal, can be used also for other purposes than formal methods.

Application of the developed formal model, the embedded language and the pattern matching extension to solve nontrivial problems of formal software development was also part of my research. I have expressed often used proof schemas of classical and temporal logic via metaprogramming and I have dealt with proving the correctness of programs using pointers, containers and iterators.

The results are summarised by three theses. The first one contains the results about the formal programming model and the embedded language. The second thesis deals with the necessary extension of pattern matching in functional languages. The third thesis shows how are the results of the first two theses applicable to solve realistic problems of formal software development.

Notions

State Space and Relational Program Model

The state space is the set of all possible states of a program, in simple cases it is the Descartes product of the sets of values of the variables used by the program. Relational program models use relations on the state space to describe the behaviour of the program or the requirements against it [15, 16]. An advantage of this method is that intuitive formulae of classical logic can be used to define subsets of the state space, and temporal logic operators can describe the program's movement between or inside these sets.

Refinement of Specification

Specification in these theses means the formal description of functional requirements that the program has to meet. If all programs fulfilling the requirements in specification $\mathcal{B}$ also fulfil the requirements in specification $\mathcal{A}$, then $\mathcal{B}$ is a refinement of $\mathcal{A}$. Refinement is the process that transforms the original specification step by step until the requirements of the most detailed specification can already be fulfilled by elementary programs. That is, this process derives the program from the specification.

Embedded Language

An embedded language is implemented by a library written in an existing programming language which is called the host language. Programs of the embedded language are in fact programs in the host language, but they are built using the entities provided by the library. The run-time environment of the host language either executes the embedded program or build up its syntax tree to be interpreted or compiled to machine code just as in case of traditional programming languages.

Pattern Matching

Pattern matching is one of the language features for branching in functional programming languages. Patterns are expressions consisting of free variables and data constructors. The first step of executing a pattern match is to compare a data to patterns. If a pattern fits the data, its free variables get bound to the respective parts of the data and the program execution continues by evaluating the branch that belongs to the pattern.

Function Patterns

Function patterns can contain functions that are not data constructors [1]. This boosts the expressive power of pattern matching, but in general it becomes indeterministic: There are function patterns that can fit a data in several different ways, so that their free variables can be bound to different sets of values. This property makes function patterns unsuitable for functional languages, while they can be applied in functional logic programming languages like Curry [14].

Separation Logic

Separation logic [17] was originally developed to describe states of programs using dynamic memory and pointers. It extends classical logic with special elementary predicates (eg. pointer p points to the value 2) and so-called separating logic operators. Separating conjunction, for example, can be used if its arguments describe disjoint parts of the dynamic memory.

Thesis 1: Embedded Language Based on a Formal Model

Results: *I have elaborated a formal program model and proved its refinement rules. Based on the model, I have defined an embedded language in Haskell that allows the derivation of programs from their specifications.*
Thesis contents: *I have generalised the known formal relational programming models for sequential [15] and parallel [16] programming.* An important feature of the model is that the instruction pointer of the program (or, in case of a parallel program, the instruction pointers of the threads) can be a component of the state space. As a result, it is possible to handle unstructured program instructions (eg. conditional or unconditional jumps) in the model. Since structured programming constructs (various branches and loops) can be built using jump instructions, it is not necessary to fix the set of control structures supported by the model: These can be added flexibly without changing the model. As a result, it is possible to keep the size of the so-called trusted base of the formal system small even in case of programming languages used in practice. This is important because the soundness of any software developed using the formal system depends on the soundness of the theory and implementation of this trusted base.

The goal of the new model, similarly to the cited models, is to enable the derivation of programs from specifications so that the resulting software is correct by construction. To achieve this goal *I have adapted temporal logic operators required to describe progress and safety criteria to the model. I have created refinement rules and have proved their soundness.* These refinement rules are not bound to concrete programming constructs and they can be used to build classical and temporal logic proofs in a uniform way.

Using first order formulae and the temporal operators of the model one can specify requirements, then use the refinement rules to transform the original specification to more and more detailed specifications until the resulting requirements are either axioms of a mathematical domain or temporal axioms of program instructions. *I have elaborated an algorithm that checks the soundness of the proofs created by this method.* In case of sound proofs the used temporal axioms determine the instructions of the program given as the result of the refinement.

I have defined an embedded language that can be used to declare variables, function and predicate symbols, describe axioms, specification and refinements. I have chosen Haskell as host language. *I have analysed Haskell's language features from the point of view of this embedding problem.* This analysis includes the representation of the entities of the embedded language, its typing and the design of its programming interface. *I have demonstrated the use of the language by specifying some elementary C++ instructions and by the derivation of simple C++ programs.*

The thesis is detailed in chapters 1 and 2 of the dissertation. The related publications are as follows. The journal paper [4] presents the formal model, its refinement rules and the verification algorithm. The journal paper [3] discusses the relation of the model to classical logic. Analysis of the embedding techniques and implementation decisions are presented in the journal paper [6]. My conference paper [2], created in international collaboration, also touches upon the topic of language embedding.

Thesis 2: Pattern Matching Extension for Embedded Languages

Results: *I have defined the notion of restricted and decidable function patterns in order to improve pattern matching in functional languages. I have shown that the basic functionality of the language extension can also be provided by a library.*

Thesis contents: The embedded language described in thesis 1 enables the implementation of heuristics that help proof construction. One needs to pattern match on entities of the embedded language (eg. formulae, expressions) to define heuristics in an easy way. Classical pattern matching is possible if the constructors of the data types are visible. However, in

case of embedded languages, constructors are usually hidden and interface functions are provided instead. These interface functions are suitable for the construction of embedded language entities, but cannot be used to deconstruct them.

Parallel to my PhD research I was also working on a project [13] as a member of an international team. This project also aims at creating an embedded language. Despite the fact that the domain of this language (digital signal processing) and the domain of the language described in the first thesis are completely different, the need for pattern matching on embedded programs became an issue also in this project. More precisely, pattern matching was needed to implement optimising transformations, but the problem outlined in the previous paragraph caused difficulties.

I have analysed in detail the available solutions of the problem described above. I have shown that the solutions using selectors, helper data types, view patterns and active patterns *are not satisfactory*. On the other hand, pattern synonyms and function patterns of the functional logic language Curry are promising. However, the expressive power of pattern synonyms is too week, while the semantics of function patterns do not fit in the classical functional programming paradigm. For this reason *I have defined the notion of restricted function patterns that is a restricted form of function patterns of the Curry language. I have defined their semantics and outlined implementation possibilities based on classical pattern matching in functional languages.* As a result, this extension does not imply run-time costs and its semantics fits in classical functional languages.

This extension is useful not only to pattern match on entities of embedded languages. *I have shown that restricted function patterns are suitable to provide list prefix patterns, $n + k$ patterns and matching on polymorphic numeric literals in a consistent way.*

I have proved that the class of restricted function patterns is undecidable. For this reason it is impossible to implement this language extension with a compiler that is guaranteed to terminate. This is unacceptable from a practical point of view. To solve the problem *I have elaborated a decision algorithm that accepts a subset of restricted function patterns and is guaranteed to terminate. I have defined decidable function patterns to be the set of function patterns accepted by the algorithm. I have shown that all the use cases enumerated above is part of this set, and that the*

decision algorithm can be built in compilers. I have defined the syntax and semantics of the language extension using a simple, purely functional base language. I have also examined how to handle undecidable function patterns.

I have elaborated a library that provides the language extension without support built into the compiler. On the other hand this solution implies run-time overhead and the programmer is in charge of checking that all the applied function patterns are restricted function patterns.

The details of the thesis are described in chapter 4 of the dissertation. The main idea of the language extension and the library providing its functionality are discussed in the conference proceedings paper [7] and in the journal paper [8]. The decision algorithm and the definition of decidable function patterns are in the conference proceedings paper [9].

Thesis 3: Solving selected problems of formal software development using language embedding and extended pattern matching

Results: *I have elaborated proof templates for certain classical logic techniques and for refinement rules of selected control structures. I have created a toolset for the formal construction of programs involving dynamic memory management, pointers, containers and iterators.*

Thesis contents: In the first thesis it was an important goal to have a model that is simple and independent of programming languages. For this reason programming constructs like specific loops and branches are not part of the model. On the other hand, one of the strength of the embedding technique used to implement the model is that the whole range of language features of the host language can be used as a metaprogramming tool-set to create embedded programs.

I have shown that metaprogramming is a safe and powerful tool to construct proofs. Using Haskell as a host language *I have elaborated proof templates for classical proof techniques* (indirect poofs, mathematical induction) *and for program derivation* (deriving rules for conditional branches and while loops). To achieve this I have used the extended pattern matching technique from the second thesis.

Proving the correctness of algorithms using pointers and dynamic memory is problematic in most of the formal systems. Separation logic in-

troduced special logical operators to express that certain formulae makes assertions about disjoint parts of the heap memory. *I have shown that my system makes the application of the effective proving style of separation logic possible in order to derive programs that use pointers and dynamic memory. I have given the axioms of the basic set of instructions to manage dynamic memory* (memory allocation and deallocation, writing and reading data via pointers). I have demonstrated the method by deriving a program that reverses linked lists in-place, by resetting pointers.

The Standard Template Library of C++ (STL) is a widely used tool. The containers, iterators and algorithms provided by the library form a convenient and extensible system, but are also sources of programming errors if the user is not experienced enough. For this reason, in case of safety critical applications, it may be important to formally derive programs that use C++ STL. As a first step towards this goal *I have elaborated a model for STL containers using the system described in the first thesis and specified the basic container operations.*

I have extended the model capable of specifying container operations in order to be able to model the connection between containers and iterators. Using the extended model *I have given the specification of the basic operations of iterators.* I have also created axioms to move between the original and the extended model within proofs.

The metaprogramming related part of this thesis is discussed in chapter 5 of the dissertation based on the journal paper [5]. Chapter 6 talks about using dynamic memory in a formally proved way. The connected publication is the conference paper [10]. Chapter 7 contains the specifications of container and iterator operations published in the conference proceedings paper [11] and journal paper [12].

References

[1] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Logic Based Program Synthesis and Transformation*, volume 3901 of *Lecture Notes in Computer Science*, pages 6–22. Springer-Verlag, Berlin, Heidelberg, 2006.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing

algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[3] G. Dévai. Programming language elements for proof construction. *Pure Mathematics and Applications*, 17(3-4):263–288, 2006.

[4] G. Dévai. Programming language elements for correctness proofs. *Acta Cybernetica*, 18(3):403–425, January 2008.

[5] G. Dévai. Meta programming on the proof level. *Acta Universitatis Sapientiae, Informatica*, 1(1):15–34, 2009.

[6] G. Dévai. Embedding a proof system in Haskell. In *Third Central European Functional Programming summer school*, Lecture Notes in Computer Science, pages 354–371. Springer-Verlag, Berlin, Heidelberg, 2010.

[7] G. Dévai. Restricted function patterns. In *Draft Proceedings of the 12th International Symposium on Trends in Functional Programming*, TFP'11, 2011.

[8] G. Dévai. Extended pattern matching for embedded languages. *Annales Univ. Sci. Budapestiensis de Rolando Eötvös Nominatae, Sectio Comutatorica*, 36:277–297, 2012.

[9] G. Dévai. Static analysis for an extension of pattern matching. In *Draft Proceedings of the 13th International Symposium on Trends in Functional Programming*, TFP'12, 2012.

[10] G. Dévai and Z. Csörnyei. Separation logic style reasoning in a refinement based language. In *Proceedings of the 7th International Conference on Applied Informatics*, pages 117–125, 2007.

[11] G. Dévai and N. Pataki. Towards verified usage of the C++ Standard Template Library. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, pages 360–371, 2007.

[12] G. Dévai and N. Pataki. A tool for formally specifying the C++ Standard Template Library. *Annales Univ. Sci. Budapestiensis de Rolando Eötvös Nominatae, Sectio Comutatorica*, 31:147–166, 2009.

[13] G. Dévai, M. Tejfel, Z. Gera, G. Páli, Gy. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, and A. Persson. Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In *Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems*, pages 12–20, April 2010.

[14] M. Hanus (ed.). *Curry: An Integrated Functional Logic Language (version 0.8.2)*, March 2006. `http://www.informatik.uni-kiel.de/~curry/report.html`.

[15] Á. Fóthi. *Introduction to programming (in Hungarian).* ELTE Eötvös Kiadó, second edition, 2007.

[16] Z. Horváth. *A Relational Model of Parallel Programs (in Hungarian).* PhD thesis, Phd School in Informatics, Eötvös Loránd University, Budapest, Hungary, 1996.

[17] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, 2002.