Introducing Certified Compilation in Education by a Functional Language Approach

Per Lindgren
Luleå University of Technology
per.lindgren@ltu.se

Marcus Lindner
Luleå University of Technology
marcus.lindner@ltu.se

Nils Fitinghoff
Luleå University of Technology
nilfit-3@student.ltu.se

Classes on compiler technology are commonly found in Computer Science curricula, covering aspects of parsing, semantic analysis, intermediate transformations and target code generation. This paper reports on introducing certified compilation techniques through a functional language approach in an introductory course on Compiler Construction. Targeting students with little or no experience in formal methods, the proof process is highly automated using the `Why3` framework. Underlying logic, semantic modelling and proofs are introduced along with exercises and assignments leading up to a formally verified compiler for a simplistic imperative language.

This paper covers the motivation, course design, tool selection, and teaching methods, together with evaluations and suggested improvements from the perspectives of both students and teachers.

1 Introduction

Software and software correctness play an undoubtedly increasing role in our society. Correctness of any software application typically relies on the correctness of the compiler at hand, where miscompilation may introduce severe and hard to find errors.

Over the last decades, formal methods have been gaining momentum in the field, see e.g., the seminal work on CompCert C[8] and LLVM verification [9]. Besides correctness guarantees, the adoption of formal methods forces rigorous semantic modelling and specifications from input language throughout the compilation process. Thus, taking the outset of formal methods into a Compiler Construction course may bring a deeper understanding of the principles of compilation techniques with additional insight into modelling and proofs of programs generally applicable to high assurance application development. So, could the concepts of certified compilation be brought into education, targeting students without prior exposure to formal methods?

A challenging task no doubt, Compiler Construction in its own right covers a vast field, and adopting formal methods without prior experience needs at least a fair introduction. At hand we have in total 9 weeks of half time studies (7.5 ECTS credits[1]), so topics covered need to be carefully selected. Additionally, exercises and labs should be designed as to maximize learning outcome and motivate students to put in the work needed to gain deeper understanding.

In Section 2 we discuss and motivate the selection of tools and teaching approach, while Section 3 details lectures and exercises. Section 4 discusses experiences gained from the first installment of the course in 2016. Here we review students' impressions and give a teacher's view. Furthermore, we discuss and detail ongoing improvements to the course (to be given fall of 2018). In Section 5 we review related work, followed by Section 6 where we summarize our contributions.

[1] European Credit Transfer System.

P. Achten, H. Miller (Eds.):
Trends in Functional Programming in Education 2018 (TFPIE'18)
EPTCS 295, 2019, pp. 65–78, doi:10.4204/EPTCS.295.5

© P. Lindgren, M. Lindner & N. Fitinghoff
This work is licensed under the
Creative Commons Attribution License.

2 Course Design

The course in Compiler Construction (D7011E) has been given on a bi-annual basis over the last two decades as an elective course to students of the Computer Science (CS) program at Luleå University of Technology. The course has established a good reputation and typically attracts some 20 students for each installment. For 2016 we decided on reshaping the course, not due to shortcomings of the D7011E course per-se but rather a lack of formal methods in the CS curriculum.

So why not a dedicated formal methods course? Well, the CS curriculum at LTU is already stacked, and introducing a new course would call for removing another. This could have been an option, we have a course on Formal Languages and Theory of Computation D7006E that interleaves with D7011E on a bi-annual basis (which traditionally attracts only a handful of students). However, bi-annual courses pose several problems. Firstly, from a teaching perspective the lack of continuity has clear disadvantages, and secondly, maintaining two interleaved courses poses twice the burden. From a student's perspective, bi-annual courses might be hard to squeeze into their studies (as there are more selectable courses than open slots in the CS program). Thus, replacing D7006E with a dedicated course in formal methods would clearly run the risk of attracting very few students (or cause migration from the D7011E course).

Hence, we opted to reshape the existing Compiler Construction syllabus from a formal methods outset and incorporate selected topics of D7006E and eventually offer D7011E on a yearly basis after phasing out D7006E from the curriculum.

2.1 Course Aims

Looking to the course aims of prior installments we find the following:

Course Aim (taken from the official course syllabus prior to 2016)
The student shall be able to

- Demonstrate the ability to identify and formulate compilation of a high-level programming language into executable machine code as a multi-phase translation process.
- Demonstrate the ability to implement a compiler for a non-trivial language using appropriate methods.
- Demonstrate the ability to present and discuss the technological solutions chosen for such an implementation in writing, in an international context.
- Demonstrate a considerable degree of specialized knowledge in the theoretical foundations of compiler technology.
- Demonstrate the competence and skill to systematically use proven tools for compiler construction.
- Demonstrate the ability to analyze and critically evaluate different aspects of modern high-level languages on the basis of their underlying implementation techniques.

Our ambition was to stay with the (already established) course aim and adopt formal methods to improve understanding on the theoretical foundations of compiler technology, while providing additional learning outcomes to high assurance programming (in and beyond the particular scope

of Compiler Construction) by introducing semantic specifications, proof techniques etc.

However, there is no free lunch, and adopting formal methods requires at least a minimum of background information on logic and methods to deductive reasoning beyond the assumed pre-requisite of discrete mathematics (M0009M, given the first year of CS studies). Moreover, in order to adopt formal methods, our input language needs to be sufficiently simple to allow for semantic modelling and proven compilation. Hence realistically, advanced language features, like object orientation, subtyping, type classes/traits, etc. will be out of reach for modelling and proofs in an introductory course.

On the upside, theoretical concepts like Structural Operational Semantics (SOS) and semantically preserving transformations (at heart of *any* language and associated compiler) becomes tangible as being concretely integrated in the students' developments. With this at hand, we have the basis for discussing extensions of a simplistic language.

In order to cover both compilation techniques and formal methods in an efficient manner, supporting tooling as well as teaching approach were carefully selected.

2.2 Tools

Firstly, a functional programming approach was selected in order to facilitate semantic modelling and reasoning on programs. Secondly, we sought a verification platform with a manageable learning curve. Among possible candidates we opted for `OCaml`[2], backed by the `Why3`[3] platform for logic reasoning and proofs of programs.

`OCaml` is a modern ML style functional language with an imperative layer and exception handling (facilitating e.g., I/O programming without introducing monads[4]). `OCaml` has a rich standard library and a plethora of supporting tools.

`Why3` is a platform for deductive program verification, where the `Why` language is used for logic (semantic) specifications, and the `WhyML` language for program implementations. `Why3` integrates to a variety of Satisfiability Modulo Theories (SMT) solvers for discharging Verification Conditions (VCs) generated for `WhyML` programs. While the process is highly automated, some degree of interaction may be required for more complex proof cases. To this end, the user can apply transformations on the generated VC(s), such as splitting, simplification, induction, etc. The `Why3` platform allows extraction of `WhyML` programs into `OCaml`, and thus provides a path for certified programming (given the assumption that the `Why3` platform and SMT solver(s) are correctly implemented).

Thirdly, we opted for the `OCaml Menhir`[5] parser generator, motivated by ease of use (well documented, good error messages, etc.). Finally we decided on the `MIPS` 3k RISC architecture as a target for code generation. In this way we could draw on the benefits of the students already being familiar with the architecture as being introduced in a prior course in Microcomputer Engineering (D0013E). Moreover, we could re-use tooling for the compiler backend (`binutils` assembler and linker) as well as the in-house development `SyncSim` for RTL emulation.

[2] https://ocaml.org/

[3] http://why3.lri.fr/

[4] For most of the students, this was their first exposure to functional/declarative programming.

[5] http://gallium.inria.fr/~fpottier/menhir/

2.3 Teaching Approach

In this course the students installed the necessary tooling on their self hosted laptops. Students brought their computers to the lectures as well as laboratory sessions. The lectures were designed to interleave the introduction of theoretical concepts with hands on demonstrations. With their laptops at hand, the students could replay the examples during class. In this way, theory and practise could be brought together, with the potential benefit of improved attention.

Laboratory assignments were designed to focus on underlying principles and concepts. In order to maximize efficiency, the students were given boilerplate code/solutions as a baseline for their own development.

Assignments were designed in an open ended fashion, with a minimal requirement for passing each associated topic (e.g., resolving parsing conflicts, construction of a virtual machine with proof of partial correctness to its SOS specification, etc.), with optional requirements towards higher grades. A high degree of freedom for extending/improving on the assignments were given, e.g., adopting more rigorous models and proofs, and/or putting efforts in implementing further code generation optimization techniques.

To further motivate the students, the "best compiler" was crowned at the end of the course.

3 Lectures, exercises and assignments

3.1 Course design and week by week agenda

For 2016, the week by week agenda was outlined as follows:

- w1 Introduction, tools, lexing and parsing.
- w2 Building your compiler frontend.
- w3 Logic and deductive program verification.
- w4 "imp" semantics and building a virtual machine (VM).
- w5 Proving correctness of the VM.
- w6 MIPS backend for "imp".
- w7 Hoare logic and verification condition generation for "imp".
- w8 Optimal and proven register assignment.
- w9 Examination (individual grading)

As seen, the agenda is quite dense, giving roughly 2 weeks for the frontend (lexing/parsing/AST generation), 3 weeks for program logic (proof techniques and verification of a VM for "imp"), and 3 weeks for the compiler backend (`MIPS` assembly code generation), encompassing register allocation and rewriting optimizations.

3.2 Lecture by lecture breakdown

Lectures were given in a fairly traditional format, each lecture being two times 45 minutes with a short break. Students were encouraged to bring their laptops to follow and replay examples throughout the lectures. In the following we outline the lecture contents as of 2016.

1. `Introduction`. In the first lecture we cover course texts (Compilers: principles, techniques and tools [1]) and Real World `OCaml` [6]), an overview of Compiler Technology (introducing concepts of lexing, parsing into AST representation, desugaring, semantic analysis, linearisation to SSA, high level optimization, RTL level optimization, and ABI conformance). A special focus was given to miscompilation and correctness (GNU C lexing [4], LLVM SSA optimizations [9]), and the CompCert C certified compiler [8]). The use of compilation techniques outside compilers were covered, such as the general use of parsers, transformations and optimizations. A first informal description of the simple imperative "imp" language was introduced.

2. `Tools`. The second lecture covers the tooling involved. The programming language `OCaml`, its package manager `opam` and the suggested Eclipse plugin `OcaIDE`; the LR(1) parser generator `menhir` (producing a shift/reduce parser in `OCaml`); and two verification platforms (`Coq` and `Why3`). In an accompanying tutorial session the students installed the tools on their own (or lended) laptops, with the assistance of a TA.

3. `Compiler Frontend`. The third lecture covers lexical analysis, regular expressions and practical details of the `menhir` parser generator. Moreover the lecture introduces EBNF grammars with examples to parse Boolean and arithmetic expressions. The concept of algebraic data types in `OCaml` is introduced. Thanks to the *extraction* of `WhyML` models to `OCaml`, the generated parser and extracted code can share the same AST definition. The lecture is accompanied with a lab assignment where the students implement new lexing rules (for `strings`). A parser for "imp" (syntax) programs into a common AST (shared with `WhyML`) is provided, however it is ambiguous. The tasks for the students are to identify conflicts and come up with a conflict free grammar with well defined precedence and associativity rules. Ambitious students are encouraged to extend the core "imp" language with additional constructs (later to be desugared into the core AST).

4. `Logic`. The fourth lecture introduces logic, and deductive reasoning. Firstly, logic inference is demonstrated on propositional logic using the `Why3` framework. Secondly, the theory of First-Order Logic (FOL) is introduced, with concepts such as *logical symbols* (quantifiers, connectives, parentheses, punctuations, and optional equality); *non-logical symbols* (predicates, functions/constants, and relations); *terms* and *expressions*; and finally *formulas* with the notion of *free* and *bound* variables with substitution under Leibniz equality, leading up to *sentences* with *truth values* and *interpretations* of FOL. From there we introduce First-order structures, with notions of *validity (Tautology), satisfiability* and *logical consequence (implication).*

   The theoretical concepts are backed with a running example of a theory with an interpretation (Peano numbers) in `Why3`. The example provides a natural outset for introducing deductive reasoning (declaring (inductive) predicates, lemmas, theorems and goals), and discussing proof approaches (e.g., proof by induction).

   The lecture is accompanied with a tutorial and a set of assignments, where the students formulate inductive predicates, learn to shape goals and devise various proofs on Peano numbers using the `Why3` framework (where proof obligations are discharged automatically using external SMT solvers).

5. `"imp" semantics`. In this lecture we give meaning (semantics) to the syntactic structure of an "imp" program. The concepts of *small-* and *big-step* Structural Operational Semantics

(SOS) are introduced along with *stores* and *configurations*. Rules for variable *lookup*, arithmetic/Boolean *reductions* (expressions), *assignments*, *sequencing*, *conditionals* and *loops* are introduced both in small- and big-step form [6].

We show how the rules for expression evaluation can be encoded in compliance to the SOS, and prove some simple reduction properties. Rules for commands in "imp" are captured in the form of an inductive predicate along with a proof of transitivity.

The lecture is accompanied by a set of assignments, where the students implement the VM, and prove partial correctness to the specification [7]. The students adopt a logic *view* of the store as a map (key/value pairs) [8]. Expression evaluation is implemented recursively over the algebraic representation, providing a well founded termination condition. However sequences and loops are not structurally decreasing. To this end, a *fueling* parameter is added, decreasing for each iteration (rendering a proof of partial correctness), where the VM is guaranteed to either return with an *out of fuel* error or a correct (final) configuration [9].

By backing the store view with a list representation, executable `OCaml` code can be extracted from the proven `WhyML` model [10].

The ambitious students are encouraged to find the appropriate fueling parameter (measure) for a given program, by tracing the SOS rules (reductions) leading up to termination. (This is a problem related to execution time analysis of the compiled program to the VM language.)

6. `Hoare Logic`. This lecture covers the generation of proof obligations (verification conditions) under Hoare logic, in a similar fashion as implemented by the `Why3` platform.

   We first review the concepts of preconditions and postconditions and their use to form Hoare triples. The axiom schemata for *SKIP*, *ASSIGNMENT*, *COMPOSITION*, *CONDITIONAL*, *WHILE* and *CONSEQUENCE* are introduced and exemplified. The *CONSEQUENCE* rule allows for strengthening the precondition and/or weakening the postcondition. Furthermore we cover the computation for weakest liberal [11] precondition (WP) for loop free code, together with an approximation of weakest precondition for *WHILE*. From there we introduce an algorithm for computing Verification Conditions (VCs), but omit formal proof of correctness.

   In order to make the WP and VC generation tangible, we encode the algorithms by *shallow* embeddings in `Why`. This allows us to demonstrate the WP/VC generation for simple programs in "imp" and discharge them as *Tasks* using the underlying SMT solvers [12]

[6] The primitive "imp" language does not support functions.

[7] Total correctness would be harder as requiring a well founded termination condition (*measure*), which is not straightforwardly obtained in the presence of loops.

[8] Variables are of unbound integer type, Booleans are constants (and cannot be assigned/stored for sake of simplicity).

[9] For any terminating program there exists a fueling parameter rendering a correct final configuration (as the "imp" language at hand does not model execution errors).

[10] Maps defined in the `Why3` standard library are theories, without backing implementation, hence useful only in proofs and cannot be extracted into executable code.

[11] Referring to partial correctness.

[12] WP/VC formulas are represented by internally generated functions, for which we cannot view the concrete representation. However, we can seek their truth values using the `Why3` platform.

7. `Stack Machine`. In this lecture we revisit the stack machine (originally defined in Double WP [5]). We review the proof approach of the compiler backend from "imp" to the stack machine language. In particular we focus on the correctness proofs for arithmetic and Boolean expression compilation, while proof details regarding the command layer are only briefly reviewed due to their complexity [2].

8. `MIPS Assembly`. In a previous course in Microcomputer Engineering, students are exposed to assembly level programming, including stack memory management. In this course they also get initial experience with compiler generated code (using the `GCC` C compiler), the compilation process and tools like `binutils ld`, and the in-house simulator `SyncSim`.

   We review the compilation and linking process and give guidelines for the use of registers. As assignments, students are to implement (unoptimized) code generation along the lines of the Stack Machine covered in a previous lecture [13]. The compiler should be able to generate assembly output, and at a minimal pass assembling, linking and testing. Ambitious students are encouraged to model the compiler in `WhyML` and use extraction to obtain executable `OCaml` code.

9. `Register Allocation`. In this lecture we review register allocation algorithms for tree expressions[3] (including optimal register allocation), and prove them correct using a similar approach to Double WP[2]. The students are encouraged to use the optimal register allocation integrated in their `MIPS` backend [14]

10. `AST level optimization`. Here we introduce a number of optimization techniques applicable at the AST level, namely for expressions *constant reduction* (e.g., $a+1+2 \rightarrow a+3$, *structural equality* (e.g., $a-a \rightarrow 0$), and *dominance* (e.g., $true||_{\rightarrow} true$), and trivial *dead code removal* for conditionals and loops at command level. The students are assigned to model the transformations in `WhyML` and prove semantic preservation to the specification. The extracted code should be integrated in the compiler and optionally enabled through an invocation parameter (`-O n`, `n` being the optimization level). This allows the students to compare generated code efficiency before and after optimization(s).

11. `General Optimization`. In this lecture we give an overview of optimization goals (speed, footprint, dynamic memory usage, power consumption etc.) and discuss obstacles and challenges with respect to correctness and complexity. Optimization at AST, RTL and post processing (link level optimization) are covered. In particular, the concept of general Control Flow Analysis with Local Optimization is discussed (constant folding/common sub-expression elimination), operator strength reduction, copy propagation, non-trivial dead code elimination. Furthermore, global optimizations based on data flow analysis are briefly covered (introducing the concepts of expressions being *defined*, *available*, and *killed*, allowing further sub-expression elimination and re-use. Moreover, code motion (hoisting), register allocation (using graph coloring) and instruction scheduling (peephole optimization and architecture aware exploitation mechanisms for pipelined and VLIW machines) are mentioned. Ambitious students are encouraged to adopt techniques (e.g. hoisting) into their working compiler.

[13] The `SyncSim MIPS` model does not implement native multiplication, thus the students have the option to generate either an error for input programs containing multiplication, or emulate multiplication (for higher marks)

[14] This can accomplished either by using extraction or by re-implementation directly in the `OCaml` compiler harness.

12. `Coq`. In the final lecture, we turn to the interactive proof assistant Coq. Here we revisit the theory of "imp" expression evaluation, and model the semantics in terms of an inductive predicate. We give a functional declaration for the evaluation as a *fixpoint* definition, and prove it by explicitly applying the induction principle (yielding two sub-goals, one for the base-case, and one for the inductive case.) While the base case can be immediately discharged (solved) by *simplification* and *reflexivity*, the inductive goal requires interactively applying various *tactics*. The example is sufficiently complex to show the trade-off between *automation* (as offered by `Why3`) and *interaction* as offered by Coq, (the later with the downside of learning curve and effort, but with the upside of a proof process, where the user is in total control provided an *informative* view of the proof state (obligations and assumptions) [15]. `Why3` allows verification conditions (including proof context) to be exported to Coq. Thus in case automatic solvers do not suffice, an interactive proof process is possible.

3.3 Exercises and Laboratory Assignments

Exercises and assignments have been designed with alignment to the learning goals in mind. Moreover, we have strived to make the assignments both motivating and challenging, with progression throughout the course, (e.g., proof techniques picked up earlier on can be later reused and refined). Moreover, students may revisit and improve on prior assignments as they mature and gain knowledge during the course.

3.3.1 Compiler frontend for "imp"

In order to facilitate the learning process, a compiler harness (`cimp` written in `OCaml`) was provided, together with a boilerplate lexer and grammar for the `Menhir` parser generator.

The compiler harness `cimp`, includes stub code for parsing command line arguments, calls the generated parser, performs basic error handling and reporting etc. In this way, the students could directly dig into working with the lexer/parser, to extend lexing rules for strings, to resolve conflicts, define precedence, and come up with their own syntactic extensions.

3.3.2 Virtual machine for "imp"

The semantic modelling was adopted from the Double WP[2, 5] `Why3` development. Double WP provides a model of a certified compiler from the "imp" core language to machine code for a minimalistic stack machine.

Here the students implemented a virtual machine in `WhyML`, and proved its correctness against the semantic specification (inductive predicate). By adopting extractable data types in `WhyML`, `OCaml` code for the proven implementation was obtained and the code could be straightforwardly integrated into the compiler harness.

3.3.3 Code transformation and `MIPS` backend

During the course we also adopted techniques from other developments such as the register allocator for tree expressions[3], and an in house development of a weakest precondition and

[15] Proof state in `Why3` is displayed as a *Task*, which may be hard to decipher and control in detail.

verification condition generation for the "imp" language (based on a shallow embedding in `Why3`).

In this exercise, the students first implemented a simplistic code generator along the lines of the *Stack Machine* covered in lectures. Here the register assignment was static, with subexpressions evaluated on the stack.

However, using the stack is highly inefficient, thus in a second part the students implemented register allocation following the lines of the optimal register allocation algorithm presented. Here, ambitious students were encouraged to implement and prove their register allocator in `WhyML`, while less ambitious students could opt for an `OCaml` implementation. As a final exercise, students were to implement and prove a set of simple code transformations at AST level. The ambitious students were encouraged to implement further optimizations (covered in lectures).

4 Results and Lessons Learned

4.1 Students' views, Course Evaluation

Each course installment is followed by a course evaluation, where the students anonymously and voluntarily fill out a questionnaire. For the 2016 installment, on average each question was answered by more than half of the students (the course was followed by 20 students in total). A summary of the poll is given below with results in italic [16]:

- How many hours of study have you on average dedicated to this course per week, including both scheduled and non-scheduled time? *A majority of the students reported 16-25h.*
- Self Assessment
  - I am satisfied with my efforts during the course. *Average 4.3 of 6.0.*
  - I have participated in all the teaching and learning activities in the course. *Average 5.0 of 6.0.*
  - I have prepared myself prior to all teaching and learning activities. *Average 3.7 of 6.0.*
- Course aims and content
  - The intended learning outcomes of the course have been clear. *Average 4.0 of 6.0.*
  - The contents of the course have helped me to achieve the intended learning outcomes of the course. *Average 5.0 of 6.0.*
  - The course planning and the study guide have provided good guidance. *Average 4.1 of 6.0.*
- Quality of teaching
  - The teacher's input has supported my learning. *Average 5.2 of 6.0.*
  - The teaching and learning activities of theoretical nature have been rewarding. *Average 4.6 of 6.0.*
  - The practical/creative teaching and learning activities of the course e.g. labs, have been rewarding. *Average 4.9 of 6.0.*

[16] The complete course evaluation amounts to 6 pages (including free text comments mostly in Swedish) can be obtained from the authors on request.

  - The technical support for communication, e.g. learning platform, e-learning resources, has been satisfactory. *Average 5.3 of 6.0.*
- Course Materials
  - The materials assigned for the course, e.g. books, lab instructions, presentation frameworks, has supported my learning. *Average 3.7 of 6.0.*
- Examination
  - The examination was in accordance with the intended learning outcomes of the course. *Average 4.0 of 6.0.*
- Overall assessment
  - The workload of the course is appropriate for the number of credits given.*Average 5.1 of 6.0.*
  - Given the aims of the course the level/difficulty of work required has been appropriate. *Average 4.9 of 6.0.*
  - My overall impression is that this has been a good course. *Average 4.9 of 6.0.*

Among the free text comments, a few highlights (translated to English) is given below:

- The course gives a good overview of what a compiler needs to do. Proof of correctness was a pleasant surprise.
- Easy to contact the teacher (Telegram) to get help/assistance. Challenging assignments.
- Good examples covered in lectures, useful to the lab assignments.
- Hands-on and theory mixed to reasonable amount. Large degree of freedom to approach and methods used in labs. Functional languages was a blast.
- Interesting subject, and interesting labs.
- Interesting setup, fun laboratory assignments, good support from teachers.
- Code generation (fun and interesting).
- Assignments should be easier to find (now as part of lecture notes).
- Change the course name to better reflect course content.
- Better distinction between lectures and tutorials (new material was brought up on the tutorials, should be moved to lectures).
- Lack of good information on Why3, somewhat fragmented feeling of lectures.
- More focus on compiler construction, less boilerplate code (ideally none).

4.2 Teacher's View

4.2.1 Overall impression

As the course was being given for the first time, we were satisfied to see that the efforts spent by the students were in line with the 20 hour per week target (the students take two courses in parallel with a total target of 40 hours workload per week). It is a delicate matter to estimate required efforts, to which end we can conclude to have succeeded. Moreover, the participation at lectures, tutorial sessions and labs remained very high throughout the course (as confirmed by the course evaluation), which leads us to believe that the students found the sessions rewarding. Additionally, the technical support using Telegram worked out very well, both as confirmed by the course evaluation and our experiences teaching the course.

4.2.2 Teaching material

At large, the teaching material (slides, lab instructions etc) sufficed to reach the course aims. However, the lack of easily accessible documentation of the `Why3` framework clearly posed some challenges. The lack of a textbook, presenting compilation techniques from a formal methods perspective was also challenging to teaching the course.

The lecture slides were designed with the aim to give sufficient background without being too verbose. For the most part, we succeeded, but there is still room for further improvements. However slides alone cannot fully replace a textbook on the subject aiming at the novice to formal methods.

4.2.3 Learning outcomes and assessment

As regarding the learning outcomes and meeting course goals, we are pleased with the overall effort spent, dedication and creativity shown by the students. Students were allowed to collaborate in groups of two for the assignments and in building a working compiler. However, most of the students chose to make individual solutions and eventually came up with their own compiler in the end.

Grading instructions were formulated as follows (snippet from the lecture notes).

Evaluation and grading
Evaluation and grading will be based on your efforts and results obtained. Grades are individual, meaning that even if you worked together on labs, be fair to each other, awarding credit where credit is due. You will bring your developments to the exam date, and demonstrate the implemented features, proofs and obtained results to defend your grade.

- Grade 3. All mandatory assignments carried out. (See per-assignment criteria)
- Grade 4. Grade 3 + documented efforts into further features and proofs. (See per-assignment criteria.)
- Grade 5. Grade 4 + documented in depth understanding and reflection of concepts covered during the course.

Since first time given in this format, no prior assessment of the efforts required is available, to this end a fair grading is only possible by evaluating Your efforts, contributions and results.

As seen, the assessment criteria for Grades 3 and 4 were quite concrete (as separately specified per-assignment), while for Grade 5 (highest) assessment was based on documented in depth understanding and reflection and thus left more open ended in order to spawn creativity.

All students were able to successfully meet the goals for Grade 3, while half of the set of students met the criteria for Grade 4 and a handful of students reached and defended the highest grade (by showing skillful adoption of theoretical concepts covered into their own developments).

4.3 What is next?

At the time of writing, the 2018 installment of the course is under preparation. We intend to keep the overall format of the course with the following set of improvements:

- Moving all development to `Why3 1.0`. The recent release offers a much improved user experience with the accompanying IDE now offering on the fly editing, interactive proof construction and improved SMT solver integration (with easily accessible counterexample support).
- Further clarification of course goals, expected learning outcomes, examination criteria and compiler competition rules.
- Split out assignments/labs from lecture material.
- Clear separation between lectures and tutorials (some restructuring of course material).
- *Strengthening the discussion of formal languages (migrated from D7006E).*
- *Introduction of fixed-width integers, bit-level operations and type checking.*

While, as one student suggested "*the compiler should be built from scratch (not based on boilerplate code)*", this is not going to happen. The reason is twofold. Firstly, the 9 week time frame does not allow for more code development. Secondly, more coding does not necessarily lead to deeper understanding of the underlying concepts. We strongly believe that boilerplate code is beneficial in the context of this type of course, as allowing (and forcing) the students to focus on the central concepts, not implementation details.

With the move to `Why3 1.0`, we expect the further streamlining of the laboratory work. Firstly, the new IDE facilitates the proof process (user interaction), and secondly we have been able to simplify the boilerplate code to completely eliminate the need for auxiliary library definitions. Furthermore, the problem of non-termination of "imp" program evaluation (and execution of VM code) are now treated by the native *diverges* contract in `WhyML`, which allow proofs of partial correctness to be achieve without introducing well founded termination conditions through additional fueling parameters. Other changes include clarifications and slide improvements, which we anticipate to further increase course efficiency.

Altogether, we feel confident that there is room to strengthen the discussion on formal languages (migrated from D7006E) and introduce fixed-width integers, bit-level operations and type checking. While the arithmetic expressions in "imp" operate on unbound integers (as reflected by evaluation and VM execution), target code for the MIPS operate either on signed or unsigned integers (both backed by 32-bit representations).

In the 2016 installment, students used signed MIPS operations causing exceptions on arithmetic overflows, thus ensuring partial correctness to the specification semantics. However, realistic languages typically provide data types and (unsigned) fixed-width operations reflecting the programmer's intent of wrapping/modulo operational semantics without overflow exceptions. Introducing (unsigned) fixed-width data types and operations thus moves the "imp" language a small but important step closer to a realistic language (in particular for high assurance embedded applications). Moreover, from a Compiler Construction point of view, by discriminating between signed and unsigned types and operations, students will encounter type checking as a step in the compilation process. For 2018 students will formulate necessary conditions for well typed programs (under the notions of type casting, conversion, and coersion), implement and (optionally) prove their type checker. Additionally, fixed-width data types enables ambitious students to add bit-vector operations (bit-wise not/and/or/xor etc.) to the "imp" language and compiler.

Obtaining tangible results is highly motivating (while obtaining proofs may not be, unless already interested in theoretical aspects). To that end, a second category for *best compiler award*

will be added, where students are competing for highest assurance compiler (most rigorously designed and proven).

5 Related/Similar Work

To the best of our knowledge, the course design is unique in introducing both compilation techniques and deductive program verification to students without prior experience in the respective fields.

The `Why3` homepage [7] lists a number of courses adopting the `Why3` platform in teaching formal methods and program verification.

- Course Proofs of Programs at the Master Parisien de Recherche en Informatique
- (in Portuguese) Courses Formal methods and Certified Programming at the Universidade da Beira Interior, Portugal
- (in French) Course Méthodes formelles et développement de logiciels sûrs at the Master Informatique de l'Université de Rennes
- (in French) Course Programmation de confiance at the Licence Informatique de l'Université de Rennes
- (in French) Course sémantique des langages, third year of Supelec Engineering School

In comparison, our focus is on the application of formal methods (not formal methods per-se). Hence, we cover theoretical concepts with less depth than would be possible in a dedicated course. Another difference is that we have a *running example* (our compiler), where we directly *apply* and *integrate* verification techniques throughout the course.

In the context of traditional courses in compiler technology, in comparison we obviously cover less ground. In particular our "imp" language is (implicitly) integer only typed at expression level. Hence, for simplicity type checking is done syntactically (at the stage of lexing/parsing). Advanced type system concepts like object orientation, Traits/typeclasses etc. are omitted from the syllabus in our case. (We aim to close the gap, by including a simple type system/typing rules for 2018.) Moreover, backend code optimization is covered in less depth than would be possible in a dedicated class.

On the other hand, in comparison to traditional compiler classes, the formalization of the structural operational semantics becomes tangible, as concretely used from (language) specification throughout the compilation process. Adopting formal methods brings the dimension of semantic reasoning, and forces the students to address problems in a rigorous manner.

6 Conclusions and Future Work

In this paper we report on our experiences introducing certified compilation techniques using a functional language approach in an introductory course on Compiler Construction. We have covered the selection of tools, teaching approach and course syllabus (including a lecture by lecture breakdown and an overview of assignments). Moreover we have discussed results and lessons learned together with an overview of related work in the field.

For the next installment (fall 2018), the course format and syllabus will remain unchanged at large. Notable improvements planned are to strengthen the coverage of formal languages, fixed-with data types and operations, type checking and add a new category of most trustworthy compiler for the student competition (in addition to the best code generation award).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman (2006): *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Martin Clochard & Léon Gondelman (2015): *Double WP : Vers une preuve automatique d'un compilateur*. In: *Journées Francophones des Langages Applicatifs*, Val d'Ajol, France, pp. 1–18. Available at `https://hal.inria.fr/hal-01094488`.

[3] Jean-Christophe Filliâtre & Martin Clochard (2013): *A tiny register allocator for tree expressions*. `http://toccata.lri.fr/gallery/register_allocation.en.html`. Online; accessed May 15th 2018.

[4] Sabine Glesner, Simone Forster & Matthias Jäger (2005): *A Program Result Checker for the Lexical Analysis of the GNU C Compiler*. *Electr. Notes Theor. Comput. Sci.* 132(1), pp. 19–35, doi:`10.1016/j.entcs.2005.01.029`.

[5] Léon Gondelman & Martin Clochard (2013): *Double WP*. `http://toccata.lri.fr/gallery/double_wp.en.html`. Online; accessed May 15th 2018.

[6] Jason Hickey, Anil Madhavapeddy & Yaron Minsky (2014): *Real World OCaml*. "O'Reilly Media, Inc.". Available at `http://www.worldcat.org/isbn/144932391`.

[7] Inria (2013): *Why3 - Home*. `http://why3.lri.fr/`. Online; accessed May 15th 2018.

[8] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:`10.1145/1538788.1538814`. Available at `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

[9] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin & Steve Zdancewic (2012): *Formalizing the LLVM intermediate representation for verified program transformations.* In John Field & Michael Hicks, editors: *POPL*, ACM, pp. 427–440, doi:`10.1145/2103656.2103709`. Available at `http://dblp.uni-trier.de/db/conf/popl/popl2012.html#ZhaoNMZ12`.