Università degli Studi di Udine

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Comunicazione Multimediale e Tecnologie dell'Informazione

Master Thesis

Retrospettiva antero-posteriore dei linguaggi funzionali imperativi

Candidate
Sara Fabro

Supervisor
Dott. Marino Miculan

Academic Year 2012-2013

Institute Contacts
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

Al mio cane,
per avermi ascoltato mentre ripassavo le lezioni.

Acknowledgements

Abstract

Contents

I Parte **1**

1 Introduction **3**
1.1 Certified Software 3
1.2 Limits of Coq Extraction 4
1.3 Thesis proposal 5
1.4 Other solutions to the problem 5

2 Haskell **7**
2.1 What is Haskell? 7
2.2 What can Haskell offer to the programmer? 8
2.2.1 Purity 8
2.2.2 Modularity 8
2.2.3 Elegant code 9

II Appendici **11**

I

Parte

1

Introduction

Developing reliable software is becoming increasingly difficult. Software's insecurity is the result of the complexity of the modern software system, the number of people involved in developing, the lack of time due to the market request, and others factors. Computer Science and Software Engineering are working at different layers with the aim to improve software's reliability: from managing software projects and structuring programming teams to programming languages and patterns to matheamtical techniques for specifying and certifying software properties.

1.1 Certified Software

Even if we don't think about it, software is involved in each aspect of our lives. Communication and computing devices, consumer products (such as cameras, refrigerators, thermostats), aviation systems, medical devices; all of them are controlled by software.

Despite this, software is still developed without a precise specification. The outcome of heedful software engineering phase is not always taken in consideration by developers, which often start to write code with an unclear or informal specification. As a result, a developer may deliver a program that will not fulfill the specification. Wide testing and debugging may contract the gap between the program delivered and the original program project, but there is no assurance that the program will satisfy the entire specification. Such inaccuracy may not always be relevant, but when joining different modules together, these approximations may lead to a system that nobody can manage or, even worse, understand. It is diffusely accepted that bugs in software are a grave problem today, since they increase the cost of software development and decrease the security of the system.

A developer that write code without a formal specification, can be compared to the mathematician describe by Bourbak, in his book: [8]

> "his experience and mathematical flair tell him that translation into formal language would be no more than an exercise of patience (though doubtless a very tedious one)"

Without a formal specification, one can occur in mistakes due to a wrong line of reasoning, like an inappropriate use of deduction or an inappropriate analogy between two different cases. The research community has developed a large number of techniques for finding bugs in programs or even proving

programs to be free of certain classes of bug. The drawback of these approaches is that they are used after the programming task, while the idea is to apply formal methods throughout the software lifecycle.

Certified software consists of an executable program C plus a formal proof P that the software is free of bugs, with respect to a particular dependability claim S. [17]. The outcome software will certify the behavior described by the claim, which can range from making almost no guarantee to simple type safety property, or all the way to deep security and correctness properties. Because the claim comes with a mechanized proof, the dependability can be checked independently and automatically in an extremely reliable way.

The conventional wisdom is that Certified Software will never be practiced because of two reasons: first, any real software must also rely on the underlying runtime system which is too low-level and complex to be verifiable; then, as a second reason, a significant part of developers asserts that the cost of a formal verification is too high compared with his benefits. In the 21st century, however, the progress and the interests in practical application of interactive theorem proving increased significantly. In the realm of pure mathematics, Georges Gonthier built a machine-checked proof of the four-color theorem [12]. His achievement is the result of six years of work, 170.000 lines of code, 15.000 definitions and 4.300 theorems. In the realm of proving verification, Xavier Leroy developed the CompCert, a verified C compiler back-end[4]. Both these two projects and many others [5] are developed and checked using the Coq Proof Assistant.

Coq is a general purpose environment for developing mathematical proofs and not a dedicated software verification tool. However, it is based on a powerful language including basic functional programming and high-level specifications. As such it offers modern ways to literally program proofs in a structured way with advanced data-types, proofs by computation, and general purpose libraries of definitions and lemmas. Coq is well suited for software verification of programs involving advanced specifications (language semantics, real numbers). [15]

1.2 Limits of Coq Extraction

As mentioned before Coq isn't a dedicated software verification tool, but it offers an extraction modul for automatic generation of programs out of Coq proofs and functions. The extraction process is based on the Curry-Howard isomorphism, which asserts that a constructive proof is isomorphic to a functional program. The main motivation for this feature is to produce certified software: each property proved in Coq will still be valid after extraction. Certified code obtained this way can be easily integrated in larger developments, so that wider communities of programmers can benefit from extraction. The languages supported by the module are three: OCaml, Haskell and Scheme.

Extraction isn't a new and revolutionary idea since it exists in Coq since 1989 and it is present in other theorem provers, like Nuprl[6]. Unfortunately, the extraction process has still one significant limit: it's all about pure functional languages. Non-functional programming languages hardly feature a type theory supporting a Curry-Howard isomorphism, and even if such a theory were available, implementing a proof-assistant with its own extraction facilities wouldn't be a valid solution. Non-functional programming languages such as C or Java do include computational effects, for example a Java function may throw an exception or modify a state during a computation.

What should a developer do in order to have certified software, with side effects, extracted directly from a Proof Assistant?

1.3 Thesis proposal

The thesis aims to extract certified Haskell code with side-effects from the Coq Proof Assistant. In order to achieve this goal, it is needed to formalize non-functional aspects. A solution to this problem is represented by the use of monads. In Computer Science, a monad describes a "notion of computation" [14]. It is a map that sends every type X of some given programming language to a new Type T(X), that is the type of T-computations with values in X. The proposal is to extend the Coq proof assistant with a suitable computation monad, that will cover the specific non-functional computational aspect. This solution is inspired by others pure functional languages, such as Haskell.

METHODOLOGY

1.4 Other solutions to the problem

Formal verification of side effect aspects is a hot topic in the Computer Science community. There are several teams and projects that are working on it, with different point of view and approach. Cybele[1] is a tool to write proofs by reflection in Coq; it is mainly developed at the Inria and the Universit Paris Diderot-Paris 7. Code is written in the Coq Proof Assistant extended with effects and termination aspects,s then procedures are extracted to OCaml. They present an innovative technique for proof-by-reflection. The key concept is the use of simulable monads, where a monad is simulable if, for all terminating reduction sequences in its equivalent effectual computational model, there exists a witness from which the same reduction may be simulated a posteriori by the monad.[9] Another project is the one conducted by Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous. They work on Formal verification in Coq of program properties involving the global state effect. [11] Their Coq framework allows to verify properties about the manipulation of the global state effect. The state doesn't appear clearly with its type of expressions which manipulate it, but it's a combination of decorations added to terms and equations. As a first result they give the proof of commutation update-update, but their main goal is to generalize to the other side effects in order to reach the verification of real-life C code with effects. The point of view changes with the approach adopted by Gabe Dijkstra.[10] His idea is to automate the process of translating Haskell code into the Coq Proof Assistant, in such way that the extracted Haskell code from Coq will have the same interface and semantics.

2

Haskell

Having a list of statements, a list of programming languages and ranking those languages in order of how well the statements apply to them, pointed out Haskell as the number one for both the following statements: "Learning this language significantly changed how I use other languages" [2] "I would recommend most programmers learn this language, regardless of whether they have a specific need for it." [3] This result isn't an unexpected surprise, since Haskell it's quite different from most other programming languages.

This chapter will give an overview of Haskell, since it is the target language of the methodology presented in this thesis. Particular attention will be given to monads and how they are embedded in the programming language.

2.1 What is Haskell?

Haskell is a static, lazy, pure functional language, based on the lambda calculus (hence the lambda in the logo), and designed over a period of three years by a group of Computer Scientists from the functional programming community. It is named after Haskell Brook Curry, an American mathematician and logician. The design effort came about because of the perceived need for a new common functional language with these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

[16]

Haskell is lazy, it won't execute or calculate a function until it's forced to determine a result. This characteristic is suitable with referential transparency, which guarantees that a function with the same

parameters always evaluates the same result in any context. The two features combined together allows to think of Haskell's programs as a series of transformations on data.

Java, C, Pascal, and many others are all imperative languages. They consists of a sequence of commands, executed with a precise and not mutable order. Haskell supports functional programming instead. The main program itself is written as a function which receives the programs input as its argument and delivers the programs output as its result. Typically the main function is dened in terms of other functions, which in turn are dened in terms of still more functions, until at the bottom level the functions are language primitives.[13] Haskell isn't simply one of many functional programming languages, it's a pure functional programming language: it does not allow any side-effect.

Haskell is a pure language; the evaluation of a program is equivalent to evaluating a function in the pure mathematical sense. Purity leads up pervasive consequences, since side effects are undoubtedly very useful. The lack of side effects was heavily perceived at the beginning, Haskell I/O was a complete and utter confusion. Necessity being the mother of invention, this shortcoming led to the invention of monadic I/O, considered one of the Haskell's main contribution to the programming world.

Haskell is statically typed, its programs are compiled before run. Its compiler will catch contingent errors at compile time, rather than finding them during the production stage. Additionally, Haskell has a good type system with type inference. As a consequence, there is no need to declare each argument's type of functions, since the type system can intelligently figure it out from its own. Type inference also allows your code to be more general. If a function takes two parameters and adds them together and it isn't explicitly stated their type, the function will work on any two parameters that act like numbers.

2.2 What can Haskell offer to the programmer?

2.2.1 Purity

As mentioned before, unlike some other functional programming languages, Haskell is pure. The result of a function is determined by its input, and only by its input; no side effects are expected. Haskell developers think about what the program is to compute not how or when it's computed, interesting even more when programming parallelize threads. On the contrary, the close relationship between imperative languages and the execution from the processor of sequencing commands implies that imperative languages can never rise above the task of sequencing. Purity it's also important because it prevents mistakes due to side effects and combined with polymorphism, it encourage a style of code that is modular, refactorable and testable.

2.2.2 Modularity

Modularity is a key concept to successful programming. When trying to write a program, one first splits the problem into sub-problems, then solves the bottom problem and tries to combine the solutions together. The ways in which one can divide the original problem is heavily influenced by the ways in which he can "glue" solutions together. A well-know analogy is the construction of a wooden chair. If the part are made separately and then glued together, the task is solved in a easy way. But if the chair has to be carved out of a solid piece of wood, it would become obviously harder. John Hughes used this

comparison in his paper:

> "Languages which aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough - modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue.
>
> Functional programming languages provide two new kinds of glue - higher-order functions and lazy evaluation." [13]

2.2.3 Elegant code

Haskell is often described as a "beautiful", "elegant" or even "cool" language, a description that is hardly associated with the committee designs for a new language. Despite that, there are some factors that have contributed to give Haskell this reputation. First of all Haskell was needed at that precise time and the goals among the committees were aligned. To understand the favourable situation in which it began to be developed, it's sufficient to cite some sentences of the Turing Award lecture delivered by John Backus in 1978:

> "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs
>
> ... An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.
>
> ...Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and are carried out in the same language in which programs are written. Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behavior and termination conditions for large classes of programs." [7]

One of the committee's priorities was the mathematical notation: clear, intuitive and elegant; to the detriment of a formally defined semantics. Many debates were stress by cries of "Does it have a compositional semantics?" or "What does the domain look like?". But in the end the absence of a formal language definition allows the language to evolve easily, because the costs of producing fully formal specifications of any proposed change are heavy, and by themselves discourage changes.

II

Appendici

Summary

Maecenas tempor elit sed arcu commodo, dapibus sagittis leo egestas. Praesent at ultrices urna. Integer et nibh in augue mollis facilisis sit amet eget magna. Fusce at porttitor sapien. Phasellus imperdiet, felis et molestie vulputate, mauris sapien tincidunt justo, in lacinia velit nisi nec ipsum. Duis elementum pharetra lorem, ut pellentesque nulla congue et. Sed eu venenatis tellus, pharetra cursus felis. Sed et luctus nunc. Aenean commodo, neque a aliquam bibendum, mauris augue fringilla justo, et scelerisque odio mi sit amet diam. Nulla at placerat nibh, nec rutrum urna. Donec ut egestas magna. Aliquam erat volutpat. Phasellus vestibulum justo sed purus mattis, vitae lacinia magna viverra. Nulla rutrum diam dui, vel semper mi mattis ac. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Donec id vestibulum lectus, eget tristique est.

Bibliography

[1] Cybele. http://cybele.gforge.inria.fr/. Accessed: 2014-04-02.

[2] Hammer principle, programming languages. http://hammerprinciple.com/therighttool/statements/learning-this-language-significantly-changed-how-i. Accessed: 2014-04-02.

[3] Hammer principle, programming languages. http://hammerprinciple.com/therighttool/statements/i-would-recommend-most-programmers-learn-this-lang. Accessed: 2014-04-02.

[4] Inria compcert. http://compcert.inria.fr/index.html. Accessed: 2014-31-01.

[5] Jscert: Certified javascript. http://http://jscert.org. Accessed: 2014-31-01.

[6] Prl project: Proof/program refinement logic. http://www.nuprl.org/. Accessed: 2014-05-02.

[7] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. 21(8):613–641, August 1978. Reproduced in "Selected Reprints on Dataflow and Reduction Architectures" ed. S. S. Thakkar, IEEE, 1987, pp. 215-243.

[8] N. Bourbaki and J. Meldrum. *Elements of the History of Mathematics*. U.S. Government Printing Office, 1998.

[9] Guillaume Claret, Lourdes Del Carmen Gonzalez Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. July 2013.

[10] Gabe Dijkstra. Experimentation project report:Translating Haskell programs to Coq programs. December 2012.

[11] Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, and Damien Pous. Formal verification in Coq of program properties involving the global state effect. In Christine Tasson, editor, *25e Journées Francophones des Langages Applicatifs, Fréjus*, January 2014.

[12] Georges Gonthier. Formal proofthe four-color theorem. *Notices of the American Math- ematical Society*, 2008.

[13] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[14] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[15] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification.

[16] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[17] Zhong Shao. Certified software. *Communications of the ACM*, 2010.