Master's Thesis Proposal "Monadic Intermediate Language for Modular and Generic Compilers"

Dmytro Lypai (900620-7113, lypai@student.chalmers.se)

January 13, 2014

The vast majority of compilers make use of different intermediate representations during the compilation process. Intermediate representation (language) is a data structure built from a source program. Classic examples are: three-address code, Register Transfer Language (RTL), static single assignment (SSA) form and many others. Compilers can use different representations of a source program during different compilation stages, depending on properties that these representations have and how suitable they are for a particular stage (for example, easy to produce from a source code, easy to transform, easy to generate code from etc.). Most of the transformations and optimisations are performed on intermediate representation(s) of a program. One of the most important and difficult considerations during different compilation stages are the effects that program fragments might cause. They have a significant influence on the kinds of transformations that may be performed.

Monads are proved to be a very powerful and generic way to encapsulate and express different effects (such as state, exceptions, non-termination etc.) as described in [1, 9]. The state of the art example is the Haskell programming language. Researchers also explored using monads and monad transformers (a way to combine different monads) for building modular interpreters and compilers [6, 5, 4, 3]. Using monads in intermediate representation enables a number of other possibilities. There are certain properties of monads (such as monad laws) that allow to perform very interesting program transformations that may influence the performance and other important properties of the resulting code significantly [2, 8].

Incorporating monads in intermediate representation for the compiler allows to describe different kinds of effects in a modular way. Different combinations of monads are suitable for programming languages with different semantics and features [7]. Building generic framework based on monads allows to express several monadic intermediate languages suitable for particular compilation stages ranging from high-level representations of source languages, where we would like to operate in terms of exceptions, computations with limited side-effects etc. to more low-level representations, which are more suitable for code generation, where the effects are mainly stores and jumps.

Therefore, the goals of the thesis are the following:

- Design a monadic intermediate language to be used as a compiler intermediate representation.
- Design a small functional programming language.
- Design a small object-oriented programming language. This part may be based on a project work done during the Compiler Construction course (TDA282).
- Build a modular and generic compiler for these programming languages using the designed intermediate representation.
- Evaluate the intermediate representation in terms of compilation of the designed programming languages.
- Evaluate how the design of the compiler allows adding new features to languages, new source and target languages.
- Evaluate the suitability of the designed intermediate representation for performing analyses and optimisations for different languages that support different programming paradigms. This is particularly important because semantic analysis and optimisations are two major parts of modern compilers.

References

[1] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. *IN INTERNATIONAL SUMMER SCHOOL ON APPLIED SEMANTICS APPSEM'2000*, 2395:42--122, 2000.

[2] Nick Benton and Andrew Kennedy. Monads, effects and transformations. *ELECTRONIC NOTES IN THEORETICAL COMPUTER SCIENCE*, 26:1--18, 1999.

[3] Laurence E. Day and Graham Hutton. Towards modular compilers for effects. In *Proceedings of the 12th International Conference on Trends in Functional Programming*, TFP'11, pages 49--64, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] William Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. *IN PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES*, pages 122--131, 1998.

[5] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. *IN EUROPEAN SYMPOSIUM ON PROGRAMMING*, 1058:219--234, 1996.

[6] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. *IN PROCEEDINGS OF THE 22ND ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES. ACMPRESS*, 1995.

[7] Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach. Bridging the gulf: A common intermediate language for ml and haskell. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 49--61, New York, NY, USA, 1998. ACM.

[8] Andrew P. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Proceedings of the Second International Workshop on Types in Compilation*, TIC '98, pages 97--115, London, UK, UK, 1998. Springer-Verlag.

[9] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24--52, London, UK, UK, 1995. Springer-Verlag.