Fully-Abstract Translations of Higher-Order Programs with Effects

Guilhem Jaber - Gallinette Team - guilhem.jaber@univ-nantes.fr

Keywords: Functional Programming - Secure Compilation - Control Operators - Type System

Ensuring that **abstraction properties** provided by a programming language are **preserved** when running a program P, written in such a language, in a potentially malicious environment is crucial in order to reason on safety of executions of P. Preservations of such properties are studied in the field of **secure compilation**, in which the so called "fully-abstract" compilation techniques are particularly appealing [PAC19].

By malicious environment, we mean an environment that may use features not present in the original programming language, in order to inspect details of the programs that are supposed to be abstracted. In the most extreme case, the program could be written in an high-level language like Java or OCaml, while the environment would be written in C or in assembly code. Then, the environment would be able to read a private field of the program, even if this is normally forbidden by the features of the programming language.

One can characterize preservation of abstraction properties using the notion of **observational equivalence** (also known as contextual equivalence). This equivalence considers the programs as black-boxed, and check that no environment can observe a distinction between them.

The goal of this project is to define a **translation that preserves observational equivalence** from an idealized higher-order (i.e. functionnal) programming language into an other one which have more observational power. More precisely, considering:

- **RefML**, a typed λ-calculus with references, that can be seen as a simple fragment of OCaml.
- Some control operator **C**, like exceptions or call/cc

then we would like to define a translation of **RefML** into **RefML+C** that preserves contextual equivalence. This means that if two programs P_1, P_2 are observationally equivalent in **RefML**, then $T(P_1), T(P_2)$ should be observationally equivalent in **RefML+C**. Even if the environments are more powerful in **RefML+C** than in **RefML**, (since they have access to the control operator **C**), they would not be able to distinguish between $T(P_1)$ and $T(P_2)$.

To define the translation T, we will explore two directions:

- using monitoring techniques [ADG$^+$15], to save (using references of **RefML**) the history of the interaction with the environment, in order to check that the interaction that the environment is performing is indeed valid. If this check fails, then the program would be allowed to raise an error.

- using some linear typing [SNRA18], as in Rust. This would allow the program to carry some kind of non-duplicable abstract resource that it would provide to the environment at each point of the interaction. The linear discipline would enforce the environment to behave in a valid way.

References

[ADG+15] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 813–830, Washington, DC, USA, 2015. IEEE Computer Society.

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, 2019.

[SNRA18] Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. FabULous Interoperability for ML and a Linear Language. In Christel Baier and Ugo Dal Lago, editors, *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume LNCS - Lecture Notes in Computer Science of *FabOpen image in new windowous Interoperability for ML and a Linear Language*, Thessaloniki, Greece, April 2018. Springer.