Ordered algebraic data types in systems programming languages

21st December 2022

Abstract Application of recent insights from logic and beautiful ideas from functional programming and linear logic to solving a long-standing problem for resource-management in modern systems programming languages (Rust/C++11): the generation of efficient *destructors* (clean-up functions) by the compilers.

Topics

- Programming languages
- Compilation
- Logic and semantics
- MPRI topics: 2.4, (2.2, 2.1)

Location Inria, Gallinette team, Laboratoire LS2N (Nantes)

Advisor Guillaume Munch-Maccagnoni <Guillaume.Munch-Maccagnoni@Inria.fr>

General presentation of the topic

A major evolution in programming languages in the recent years has been the advent of modern *resource-management* features in the systems programming languages C++11 (Stroustrup et al., 2015) and Rust (Matsakis and Klock II, 2014). In what one might initially think unrelated, many recent advances in the theory of programming languages are due to the *Curry-Howard correspondence* between functional programming, logic and category theory.

In our team, we have been studying some of these systems programming features, via the Curry-Howard correspondence, from the point of view of *linear logic* (Girard, 1987; Melliès, 2009; Baker, 1994), and established a connection with a variant called *ordered logic* or *non-commutative logic* (Lambek, 1958).[1] This opened further research directions on the general theme of merging of systems programming and functional programming (with the ultimate goal of creating the Next Best Programming Language).[2]

Of these research directions, some can interest future young researchers who are looking for a hands-on approach to programming language development, others can interest more mathematically-inclined ones who are interested in advancing the theory of programming languages and its connection to logic. This internship subject can interest both and can evolve into a PhD subject in either direction.

[1] Combette and Munch-Maccagnoni, 2018.
[2] Munch-Maccagnoni, 2018.

Goal of the internship

Types with destructor Recently, we have proposed the notion of *ordered algebraic data types* to model the types of resources in C++11/Rust: those types with an associated *destructor,* a clean-up action (such as freeing some memory, closing a file...) that has to be called reliably and predictably at the end of the value's lifetime.[3] This is the primary mechanism for dealing with memory in these languages without a garbage collector.

Ordered algebraic data types are algebraic data types (like in OCaml) to which a destructor is associated, in a way that follows structurally from the type. For instance, when A and B have associated destructors (for instance if they have been defined by the user with an explicit effectful destructor), the type of pairs $A \otimes B$ has a destructor which first calls the destructor of A and then the one of B. The types $A \otimes B$ and $B \otimes A$ are distinct, because they do not destroy A and B in the same order, hence the adjective *ordered*.

There are for instance two types of list: $\mu X.(1 \oplus (A \otimes X))$ defines the type of lists whose elements are destroyed in list order, whereas $\mu X.(1 \oplus (X \otimes A))$ defines the type of lists whose elements are destroyed in reverse order.[4] In Rust, this corresponds to the following two types which differ similarly in the way they dispose of their values:

```
pub struct List<A> { node: Option<Box<(A, List<A>)>>, }
pub struct List2<A> { node: Option<Box<(List2<A>, A)>>, }
```

For such types defined by the user, the destructor is automatically generated by the compiler based on whatever destructor is associated to type parameter A.

The stack overflow issue It is a known issue in C++11 and Rust that the naive destructor implemented recursively (as currently done by all compilers currently) causes a *stack overflow* (does too many recursive calls) on too deep data. The user is then encouraged to reimplement the destructor by hand, sometimes trading *stack space* for time (by traversing repeatedly) or for *heap space* (by enqueuing elements to be destroyed), and sometimes by altering its meaning (changing the order or delaying the destruction). It was widely believed that the ideal destructor did not exist and thus could not be generated by the compiler.

In M.-M. and Douence (2019), we showed how for ordered algebraic data types, the correct destructor could be implemented with constant stack and heap space usage, and how it could be derived in a systematic manner. It combines and extends two beautiful ideas developed in functional programming. (It can be read for a more detailed contextualisation, as it is quite short.)

Scaling the solution We want to make this theoretical result applicable, for instance as a concrete proposal in discussions with Rust developers. For this purpose, it is necessary to extend the result to *abstract data types*. Abstract data types are those whose definition is hidden from the user, and not known in advance to the compiler (this includes library-defined container types such as vectors). In practice, types found in programs are composed from such container types (e.g. **Box** in the Rust example), while the method we have developed so far only works on fully-known types. This extension seems doable and interesting.

Internship task The internship task is to understand the paper (Munch-Maccagnoni and Douence, 2019) with the help of the advisor, be able to present it in more details, and, as a research in team

[3] Munch-Maccagnoni and Douence, 2019.

[4] Using analogies between functional programming and linear logic that are important here, we write $\otimes$ for the constructor of pairs and $\oplus$ for the constructor of variant type (also called sum types). μ denotes a least fixed-point.

with the advisor, to extend the result as explained above, and, if successful, to contribute to the publication of the result in an international conference or journal.

According to the student's taste, the development can remain purely theoretical, or it can be supported by a prototype implementation in OCaml, or a formalisation of proofs in the Coq proof assistant (the student interested in the Coq proof assistant will benefit from the Gallinette team's expertise on Coq).

Notes

- Our team Gallinette provides a nice atmosphere. It is a large and young team of researchers in Nantes, specialised in logic, programming languages, and the formalisation of mathematics. Many colleagues and students work on the development and application of the Coq proof assistant.
- My speciality is in logic and denotational semantics (e.g. lambda-calculus). More recently, I have found this area of application of my works to programming languages[5], and I started hacking and contributing to the OCaml language runtime. I interact both with researchers in logic/semantics and with OCaml/Rust researchers and implementers in the industry.
- Funding is available for travel (e.g. workshops and visits abroad). Collaborations are possible with the University of Cambridge.

Expected abilities of the student As a young researcher, you probably already have:

- outstanding creativity,
- taste and capacity for acquiring a bibliographic knowledge of a topic,
- teamwork,
- clear and rigorous writing,
- good oral presentation skills.

More specifically for this topic, and less importantly, one expects:

- Some interest in programming languages, their theory and implementation, or the Curry-Howard correspondence.
- Knowledge of, or interest, in learning Rust (or C++) at a basic level.
- Optionally some experience in, and taste for, programming in OCaml.

References

Henry G. Baker. 1994. Linear logic and permutation stacks - the Forth shall be first. *SIGARCH Computer Architecture News* 22, 1 (1994), 34–43. https://doi.org/10.1145/181993.181999

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. *A resource modality for RAII (abstract)*. Technical Report. INRIA. https://hal.inria.fr/hal-01806634

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

Joachim Lambek. 1958. The mathematics of sentence structure. *The American Mathematical Monthly* 65, 3 (1958), 154–170.

Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[5] See for instance my presentation of this topic at the Collège de France: https://www.college-de-france.fr/site/xavier-leroy/seminar-2018-12-19-11h30.htm.

Paul-André Melliès. 2009. *Categorical semantics of linear logic*. Panoramas et Synthèses, Vol. 27. Société Mathématique de France, Chapter 1, 15–215.

Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. (2018). https://arxiv.org/abs/1803.02796

Guillaume Munch-Maccagnoni and Rémi Douence. 2019. *Efficient Deconstruction with Typed Pointer Reversal (abstract)*. Technical Report. INRIA. https://hal.archives-ouvertes.fr/hal-02177326

Bjarne Stroustrup, Herb Sutter, and Gabriel Dos Reis. 2015. A brief introduction to C++'s model for type- and resource-safety. (2015). http://www.stroustrup.com/resource-model.pdf