Towards Reasonable Ownership

Anya Helene Bagge[1], Kristoffer Haugsbakk[1], and Vadim Zaytsev[2]

[1]University of Bergen, Norway [2]Raincode Labs, Belgium

Introduction

Mutable state is a feature useful for programmers, but its combination with sharing or aliasing makes programs much harder to reason about, both for humans and automated tools for analysis, refactoring and optimisation. In this short paper we explore options for managing sharing and dataflow to simplify automated rewriting of programs. Consider the following code fragment:

```
List<Employee> list = ....;
Company c = new Company(list);
list.sort();
c.doSomething();
if (list.isSorted()) // is the list still sorted?
```

Assuming Java-like semantics, the exact behaviour of this code depends on the implementation of at least `Company`, and possibly other parts of the program. The list may be stored as a reference inside the company object, allowing it to manipulate the list, which may become unsorted during the `doSomething` call. However, if we are assured that no aliasing occurs, or that no statement modifies the list between the call to `sort` and the `if`, we (both the programmer and a hypothetical automated tool) can infer that the list is guaranteed to be sorted, and we may optimise the `if`-statement.

Sufficient knowledge of situations possible during execution of each statement, enables many things: some refactorings only work on alias-free code (and their practical implementations leave the safety questions to the programmer [5]); many optimisations reorder or parallelise statement executions; interprocedural analyses become simple; the statement form (update-oriented) and the expression form (value-oriented) of the same code get freely interchangeable [2].

Magnolia

We started experimenting with options listed above, in the Magnolia programming language [1,2]. Its rules against aliasing are strict and control the flow of data in and out of procedures through *parameter modes*: `obs`[erve] — the parameter is read, but never modified; `upd`[ate] — both read and written to; `out`[put] — write-only. From these, we can tell which objects the result of a procedure call depends on (the set of `obs` and `upd` arguments), and which object can be changed by a procedure call (the set of `upd` and `out` arguments). Aliasing is forbidden through the following strict rules: (1) assignment is copying; (2) references are only used parameter passing; (3) an `upd`/`out` variable may not occur

more than once in a parameter list. The Magnolia design is not unlike that of *fluent languages* [6] that combine functional and imperative styles by separating the language into sublanguages, with *procedures* which are fully imperative, *observers*, which do not cause side-effects, *functions*, which do not cause side-effects and are not affected by them, and *pures* which are referentially transparent (no side-effects, and return the same value on every evaluation). The invariants are maintained by forbidding calls to subroutines with more relaxed restrictions.

Rust and Metal

Rust [8] is a systems programming language, designed for typical low-level tasks like implementing OS kernels. Unlike most low-level languages, it is memory safe as long as escape hatches are not used. Rust uses affine types and an ownership system which makes sure statically that safety-ensuring invariants are not violated. The escape hatch is the `unsafe` construct which delegates the task of ensuring the invariant checks for each given unsafe block, function, trait or implementation, to the programmer.

Metal [3] is a Rust-based language with static tracking of ownership and preventing aliasing. Ownership there is managed implicitly, as opposed to in Rust where one often needs to annotate functions with so-called *lifetimes* which are associated with the types. Unlike Rust, Metal can track locations with greater precision. However, Metal lacks a lifetime system, which means that it cannot implement things like Rust's `Drop` trait (a trait in the standard library implementing *object destructuring* in a hierarchical LIFO order).

Rust uses pointers and affine types to avoid aliasing of those pointers. In contrast, Magnolia has no pointers so there is no aliasing to avoid in the first place. Rust uses *algebraic data types* from functional languages like ML, which makes it especially simple to deal with sum types compared to in C and OO languages. Magnolia deliberately uses *abstract* (sic!) *data types*, which do not have to expose their structure and allow more implementation freedom.

As a matter of principle, Rust makes all potentially costly constructs explicit. Thus, it uses *move-by-default* for assignment, which is unlike both C++ and Magnolia. This makes the use and transfer of ownership more ergonomic, and copying assignments more explicit. However, values that are cheap to copy, are not moved, and the programmer is allowed to make such distinctions by using certain traits. This is an improvement on the model of C++, which also aims to provide *zero-cost abstractions* to the programmer. Compared to Magnolia, Rust provides more of a conceptual burden on both the code reader (more explicit information) and the library author (more traits to check).

In general, Rust provides no tools for checking foreign code, and is usually meant to interoperate with C code which has a similar memory model. Magnolia is very different since its specifications work the same for foreign and native code—it expects any foreign code to be well-behaved. It will enforce no-alias rules in calls to foreign code, but aliasing, pointers and non-determinism must be hidden behind the interface presented to Mangolia.

Applied Type System

ATS is a software language that combines programming with theorem proving [4] with both dependent types and a form of linear types. In contrast to theorem provers like Coq [9] and NuPrl [7], ATS does not employ *program extraction* in order to derive verified programs, and proofs are guaranteed to be erased before runtime [4]. It achieves this by having a clear separation between proofs and programs. One of the ideas behind ATS is that programming based on Martin-Löf's constructive type theory seems to be too restrictive for general purpose programming. In particular, the resulting type systems require that programs are pure and total, while realistic programs require things like general recursion, pointers and exceptions [4]. To bridge the gap, ATS provides only limited integrated combination of program construction and verification. That puts it somewhere between Coq and Magnolia which does not integrate programs and verification as tightly as ATS, allowing foreign code to implement specifications and still be able to leverage Magnolia's specification and verification facilities, such as axioms.

Memory-safe programming often necessitates certain restrictions, one of them being pointer arithmetic. In contrast, ATS allows pointer arithmetic by providing the programmer with tools for proving the safety of the pointer manipulation through *stateful views* [11]. We can compare this to Rust, which does not allow pointer arithmetic in its safe subset. (Strictly speaking, Rust allows pointer arithmetic on so-called *raw pointers* in its safe subset, but such pointers are not allowed to be dereferenced outside of `unsafe` blocks.) The programmer has to put such things in `unsafe` blocks and verify by herself that the code maintains the invariants required of safe code. One can also use stateful views to prove that arrays are not indexed out of bounds. Arrays are an important data structure in general in programming [10], and especially in Magnolia, since arrays provide efficient implementation for matrices on many architectures, which is important for the numerical domain that Magnolia is geared towards. So, both ATS and Magnolia provide for efficient data structures like arrays: ATS allows for proving properties about indexing of arrays, while in Magnolia one uses axioms to specify the range of arrays.

Conclusion

Control of aliasing and the flow of data in and out of procedures present us with interesting opportunities for reasoning about programming. Alias protection is a prerequisite for reliable information on data flow. With ownership types we may achieve this even without onerous restrictions on sharing, mutability and references. Several angles may be interesting to explore further. Can we achieve fine-grained effects control with algebraic properties? Could we use ownership without static data types? How simple can the ownership type system be, while still providing reasoning benefits?

References

1. Bagge, A.H.: Constructs & Concepts: Language Design for Flexibility and Reliability. Ph.D. thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway (2009), http://www.ii.uib.no/~anya/phd/
2. Bagge, A.H., Haveraaen, M.: Interfacing Concepts: Why Declaration Style Shouldn't Matter. In: Ekman, T., Vinju, J.J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09). vol. 253, pp. 37–50. Elsevier (2010)
3. Benitez, S.: Rusty Types for Solid Safety. In: Proceedings of the Workshop on Programming Languages and Analysis for Security. pp. 69–75. ACM (2016)
4. Chen, C., Xi, H.: Combining Programming with Theorem Proving. In: ACM SIGPLAN Notices. vol. 40, pp. 66–77. ACM (2005)
5. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer Refactorings. In: Margaria, T., Steffen, B. (eds.) Proceedings of the Seventh International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16), Part I. pp. 517–531. Springer (2016)
6. Gifford, D.K., Lucassen, J.M.: Integrating Functional and Imperative Programming. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 28–38. LFP '86, ACM (1986), http://doi.acm.org/10.1145/319838.319848
7. Kreitz, C.: Nuprl as Logical Framework for Automating Proofs in Category Theory. In: Constable, R.L., Silva, A. (eds.) Logic and Program Semantics: Essays Dedicated to Dexter Kozen on the Occasion of his 60th Birthday. pp. 124–148. LNCS, Springer (2012)
8. Matsakis, N.D., Klock, II, F.S.: The Rust Language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 103–104. HILT'14, ACM (2014)
9. The Coq Development Team: The Coq Proof Assistant Reference Manual, Version 8.6. https://coq.inria.fr/distrib/current/refman/ (2016)
10. Zaytsev, V.: Language Design with Intent (2017), submitted to the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2017). The card on arrays is called http://slebok.github.io/dyol/cards.html#Collection.
11. Zhu, D., Xi, H.: Safe Programming with Pointers through Stateful Views. In: International Workshop on Practical Aspects of Declarative Languages. pp. 83–97. Springer (2005)