Programme for a rational reconstruction of ownership in PLs

Guillaume Munch-Maccagnoni

ProgLang@Inria meeting, Paris, February 8th 2023[1]

[1]: Updated from feedback, Feb. 21st.

Introduction

- About resources in programming languages (an abstraction to reason about state)
- How to gain further understanding of it via models in denotational semantics
- Challenges (technical, methodological)

Introduction

Goals

- Present a set of research questions in denotational semantics motivated by language design problems
- Locate this effort within an approach to PLs that mixes data-gathering from the real world, and a critical view of the relationship between semantics and programming

Ownership/Uniqueness

Control of aliasing

```
# let m = Array.make 4 (Array.make 4 0);;
val m : int array array =
 [|[|0; 0; 0; 0|];
   [|0; 0; 0; 0|];
   [|0; 0; 0; 0|];
   [|0; 0; 0; 0|]|]
# m.(0).(0) <- 128;;
- : unit = ()
# m;;
- : int array array =
[|[|128; 0; 0; 0|];
  [|128; 0; 0; 0|];
  [|128; 0; 0; 0|];
  [|128; 0; 0; 0|]|]
```

Ownership/Uniqueness

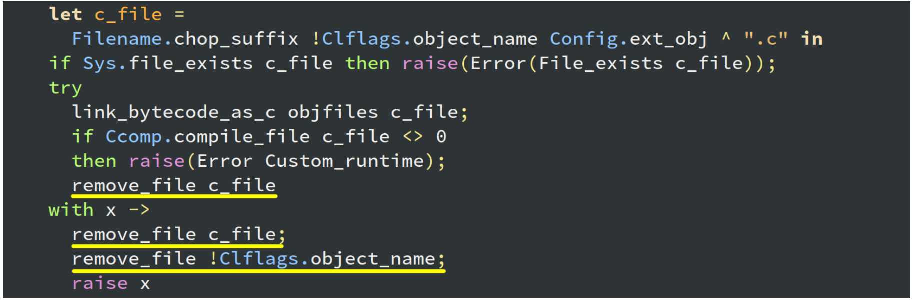
Control of aliasing

Control of aliasing

- Reasoning about state (cf. iterator invalidation)
- Concurrent programming (ownership transfer & other patterns of non-interference)
- Optimizations (C `restrict`; memory re-use)

Ownership/Uniqueness

Resource management (bytecomp/bytelink.ml @ 8f58956 (Nov. 1996))

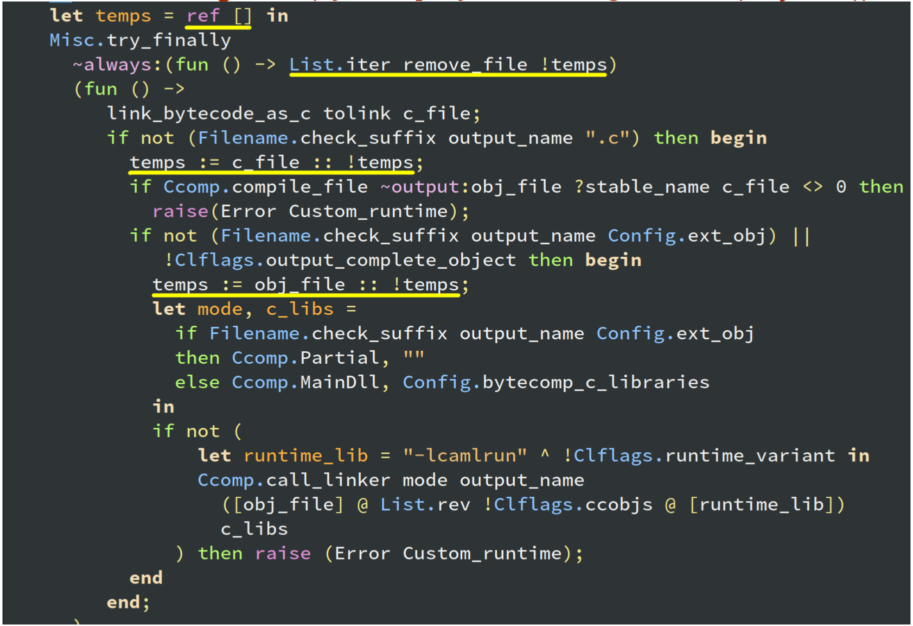


```
let c_file =
  Filename.chop_suffix !Clflags.object_name Config.ext_obj ^ ".c" in
if Sys.file_exists c_file then raise(Error(File_exists c_file));
try
  link_bytecode_as_c objfiles c_file;
  if Ccomp.compile_file c_file <> 0
  then raise(Error Custom_runtime);
  remove_file c_file
with x ->
  remove_file c_file;
  remove_file !Clflags.object_name;
  raise x
```

Note: example found by systematic audit of patterns of resource-management in the OCaml compiler implementation

Ownership/Uniqueness

Resource management (bytecomp/bytelink.ml @ 40bab2d (July 2018))



```
let temps = ref [] in
Misc.try_finally
  ~always:(fun () -> List.iter remove_file !temps)
  (fun () ->
     link_bytecode_as_c tolink c_file;
     if not (Filename.check_suffix output_name ".c") then begin
       temps := c_file :: !temps;
       if Ccomp.compile_file ~output:obj_file ?stable_name c_file <> 0 then
         raise(Error Custom_runtime);
       if not (Filename.check_suffix output_name Config.ext_obj) ||
          !Clflags.output_complete_object then begin
         temps := obj_file :: !temps;
         let mode, c_libs =
           if Filename.check_suffix output_name Config.ext_obj
           then Ccomp.Partial, ""
           else Ccomp.MainDll, Config.bytecomp_c_libraries
         in
         if not (
             let runtime_lib = "-lcamlrun" ^ !Clflags.runtime_variant in
             Ccomp.call_linker mode output_name
               ([obj_file] @ List.rev !Clflags.ccobjs @ [runtime_lib])
               c_libs
           ) then raise (Error Custom_runtime);
       end
     end;
  )
```

Ownership/Uniqueness

Resource management

Resource management

- Memory management (malloc/free)
- Typestate/protocols
- Interoperability
- Fault tolerance (exception handling)

Ownership/Uniqueness

The Rust programming language represents a breakthrough for all these questions

- C++11 (RAII (destructors) + move semantics, among many other things): above (at an industrial scale, + structure)
- Type system for ownership & borrowing (systems programming/OOP + "linear borrows")

(Matsakis and Klock II, 2014; Anderson et al., 2016)
Like C++, it arose outside of academia

Ownership/Uniqueness

Approaches in this area

- Linear type systems: type systems that count how many times a variable appears
  (Wadler, 1991, and others)
- Program logics, e.g. separation logic: quite successful in verifying non-toy systems including Rust
  (Reynolds, 1978; O'Hearn et al., 1999, and others)
- Ownership type systems (OOP & systems communities): greater focus on language design, more clearly a source of inspiration for Rust
  (Clarke and Wrigstad, 2003; Jim et al., 2002, and others)

Rational reconstructions

Rational reconstructions

- Build an understanding via a refined (=épuré) model where features stand by themselves
- Connecting with existing bodies of knowledge (e.g. λ-calculus and its semantics as a bridge between intuitionistic logic and functional programming)
- Opinionated theories (not some program logics that you could apply to any programming language good or bad)

Rational reconstructions

Example: continuations

Continuations: Historically lots of different approaches

- Semantics: categorical (monad, comonad), translations (CPS, Gödel-Gentzen, into linear logic)
- Many (!) different formalisms
- Many different questions: programming (control operators), logic (classical translations)

Rational reconstructions

Example: continuations

Rational reconstructions:

- Girard (1991), Danos et al. (1997): a logic that generalizes all (many) approaches
- Thielecke (1997), Levy (1999) connecting with the study of effects
- Curien and Herbelin (2000): idem for syntaxes/calculi
- Melliès: building blocks that one composes (Melliès and Tabareau, 2010)

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend left, "L"] \arrow[r, phantom, "\perp"] & \mathscr{L} \arrow[l, bend left, "M"] \arrow[r, bend left, "\mathrm{op}"] \arrow[r, phantom, "\perp"] & \mathscr{L}^{\mathrm{op}} \arrow[l, bend left, "\mathrm{op}"] \arrow[r, bend left, "M"] \arrow[r, phantom, "\perp"] & \mathscr{C}^{\mathrm{op}} \arrow[l, bend left, "L"]
\end{tikzcd}$$

Rational reconstructions

Linear call-by-push-value

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend left, "L"] \arrow[r, phantom, "\perp"] & \mathscr{L} \arrow[l, bend left, "M"] \arrow[r, bend left, "\mathrm{op}"] \arrow[r, phantom, "\perp"] & \mathscr{L}^{\mathrm{op}} \arrow[l, bend left, "\mathrm{op}"] \arrow[r, bend left, "M"] \arrow[r, phantom, "\perp"] & \mathscr{C}^{\mathrm{op}} \arrow[l, bend left, "L"]
\end{tikzcd}$$

- Linear call-by-push-value (2016): how to combine *resource modalities* and *effect modalities*
- Girard: Logic of Unity (1993). Mix linear & non-linear continuations (Discussed recently: how to add duplicable continuations to OCaml?)

Rational reconstructions

A resource modality for RAII

- Linear Call-by-push-value (2016): combination of *resource modalities* and *effect modalities*
- Combette & M. (2018). Connection between types with destructors and ordered logic.

$$\begin{tikzcd}
\mathscr{V}/TI \arrow[r, bend left] \arrow[r, phantom, "\perp"] & \mathscr{V} \arrow[l, bend left, "\square"] \arrow[r, bend left, "\lozenge"] \arrow[r, phantom, "\perp"] & \mathscr{V}^{T(-\oplus E)} \arrow[l, bend left]
\end{tikzcd}$$

Rational reconstructions

A resource modality for RAII

- A type-based abstraction. Attach a destructor to a type, to create a new type.
- *Ordered* data types (rather than linear or affine)

$$A \otimes B \not\cong B \otimes A$$

- Still affine at the level of provability!

$$A \otimes B \leftrightarrow B \otimes A$$

- Solves open question of combining linearity and control effects (with lots of thanks to C++ RAII)

$$\Diamond A \to \Box(A \to \Diamond B) \to \Diamond B$$

"One needs to know how to discard a computation in order to propagate an exception"

Rational reconstructions

A resource modality for RAII

“Are types in Rust linear or affine?”
Our model is clear:

- *Linear* at the level of values
- *Ordered* at the level of types
- *Affine* at the level of provability

Rational reconstructions

A resource modality for RAII

$$\mathrm{List}(A) = \mu X.(1 \oplus (A \otimes X))$$

$$\mathrm{Tsil}(A) = \mu X.(1 \oplus (X \otimes A))$$

Rational reconstructions

A resource modality for RAII

$$\mathrm{List}(A) = \mu X.(1 \oplus (A \otimes X))$$

$$\mathrm{Tsil}(A) = \mu X.(1 \oplus (X \otimes A))$$

- The stack overflow issue
  - Open problem in C++, Rust, Swift...
  - Typed pointer reversal (solution from functional programming)

Research questions

ML with resources?

How to add first-class resources to ML?

Mix several resources and effects in the same language

$$\begin{tikzcd}
\mathcal{O} \arrow[dr, bend left, "\&"] \arrow[dr, phantom, "\perp"] & & \\
& \mathcal{V} \arrow[ul, bend left] \arrow[dl, bend left] \arrow[r, bend left] \arrow[r, phantom, "\perp"] & \mathcal{C} \arrow[l, bend left] \\
\mathcal{M} \arrow[ur, bend left] \arrow[ur, phantom, "\perp"] & &
\end{tikzcd}$$

```
type t = Res u with destructor f
&t
```

e.g. Kind system inspired by polarities (Girard, 1991, 1993).

- *Qualitative* linearity (e.g. special traits in Rust), as opposed to quantitative linearity (counting how many times variables are used)
- Nevertheless expected to be compatible with lessons from affine type systems

Research questions

Types of closures

Reconstruct what we already know
Example: types of closures

$$A \rightarrow_p B \overset{\text{def}}{=} \Box_p(A \rightarrowtriangle B) \qquad (p \in \{\mathrm{M}, \mathrm{O}\})$$

- The kind of a function does not depend on inputs and outputs
- Distinction between functions and closures
- Different kinds of closures (depending on what is in the closure)
- In Rust: Fn, FnOnce, FnMut

Research questions

Types of closures II

Making predictions

Tov and Pucella (2011): practical affine types (kind system with principal kinds)

$$t \rightarrow_{\langle\alpha\rangle} u \qquad (\langle\alpha\rangle \in \{\mathrm{A},\mathrm{U}\})$$

- We do not reconstruct such a refined type system…
- …but, they have noticed that currified functions tend to accumulate annotations in a predictable manner

$$\forall\alpha\beta(\alpha \rightarrow \beta \rightarrow_{\langle\alpha\rangle} t \rightarrow_{\langle\alpha\rangle+\langle\beta\rangle} u)$$

The model predicts a way by which by introducing explicitly a primitive ("call-by-push-value") arrow, one can remove superfluous annotations

$$\forall\alpha\beta(\alpha \rightarrowtriangle \beta \rightarrowtriangle t \rightarrowtriangle u)$$

(see also the treatment of currying in F#)

Research questions

Rational reconstruction of ownership

Challenges to test the model

In Rust/C++, linearity and ownership are *emergent phenomena* of types with destructors (resource types/ownership types).

Other notions follow intuitively from them in Rust:

1. Region typing (“borrows”),
2. Uniqueness (“linear borrows”),
3. External uniqueness/linear abstract data types (“interior mutability”).

Can this intuitive hierarchy be explained in denotational semantics?

Research questions

Rational reconstruction of ownership

Challenges to test the model

What is borrowing? How does it appear?

- Hypothesis: “&” as forgetful functor from ownership types to the base category (linear/copiable)

$$\&(A \otimes B) = \&A \otimes \&B$$

How does it prevent use-after-free if the result of a borrow is a copiable type?

- Related to a programming problem: how can I define resources starting from types all copiable?
- Hypothesis: mix of kind system + destructors + borrowing + linear abstract data types

$\Rightarrow$ Methodological limits to the “toy system” approach

Research questions

Rational reconstruction of ownership

Other open problems interesting to look at from this angle

- Erlang/Rust panic model
    - Ahman and Bauer (Ahman and Bauer)
- Limitations of Rust borrow checker
    - Revisit type-and-effect systems for ownership

Challenges in language design

As you might have noticed

- Intertwined considerations from logic to computer architecture
- Requires lots of knowledge about the diverse problems faced by programmers
- Diminishing returns of the experience of writing compilers
- Limitations of the “toy language” approach
- There is more to science than making a falsifiable claim (such as type safety)
- Formal methods: how do you take into account emergent code and reasoning patterns? (cf. resource-management example at the start)

Possible keys

This approach:

- A critical view of Curry-Howard (see e.g. *"the Romance of Mathematics"* about monads in Petricek, 2018)
- Allows more distance between model/toy formalism and language proposition, requiring a rational (not necessarily technical) discourse to connect to programming languages
- Responsibility for the "owners" of the means of production of knowledge (e.g. languages with critical mass to gather user feedback and experience)
- Go back at the roots of our belief in mathematical approaches (e.g. Priestley, *"The Algol Research Programme"*, 2011.)

Possible keys

Structured programming

Structured programming (Dijkstra)

- Correctness should follow from the structure of the program
- The structures provided by the programming language should facilitate reasoning about the program

(Priestley, 2011)

Possible keys

Structured programming

Structured programming (Dijkstra)

- Correctness should follow from the structure of the program
- The structures provided by the programming language should facilitate reasoning about the program

(Priestley, 2011)

> *"[Destructors are] a systematic approach to resource management with the important property that **correct code is shorter and less complex** than faulty and primitive approaches. [. . .]*
>
> *The introduction of exceptions [...] was delayed for about half a year until I found "resource acquisition is initialization" as a **systematic and less error-prone** alternative to the finally approach."*

(Stroustrup, 2007, emphasis mine)

Possible keys

C++ as a 40-year-long experiment

“C++ is built on the idea of incremental growth and the gradual replacement of older facilities with newer ones where appropriate.” (Stroustrup, 2020)
(Rust follows a similar approach.)

Possible keys

C++ as a 40-year-long experiment

“C++ is built on the idea of incremental growth and the gradual replacement of older facilities with newer ones where appropriate.” (Stroustrup, 2020)
(Rust follows a similar approach.)

A theory of programming language design and evolution

- rooted in the socio-technological context of programming languages,
- rooted both in experience *and* (to my initial surprise) the overarching research programme of our community,
- that seeks relative claims (within one language), where one cannot find evidence for absolute ones (between all languages).

Conclusion

Thank you

References I

Danel Ahman and Andrej Bauer. Runners in Action. In *Programming Languages and Systems* (2020), Peter Müller (Ed.). Springer International Publishing, 29–55.

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *ICSE '16*. https://doi.org/10.1145/2889160.2889229

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

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science)*, Luca Cardelli (Ed.), Vol. 2743. Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

References II

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

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *Proc. POPL*. https://doi.org/10.1145/2837614.2837652

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. *ACM SIGPLAN Notices* 35 (2000), 233–243.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A New Deconstructive Logic: Linear Logic. *Journal of Symbolic Logic* 62 (3) (1997), 755–807.

Jean-Yves Girard. 1991. A new constructive logic: Classical logic. *Math. Struct. Comp. Sci.* 1, 3 (1991), 255–296.

Jean-Yves Girard. 1993. On the Unity of Logic. *Ann. Pure Appl. Logic* 59, 3 (1993), 201–217.

References III

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. `http://www.usenix.org/publications/library/proceedings/usenix02/jim.html`

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Proc. TLCA '99*. 228–242.

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

Paul-André Melliès and Nicolas Tabareau. 2010. Resource modalities in tensor logic. *Ann. Pure Appl. Logic* 161, 5 (2010), 632–653.

Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. 1999. Syntactic Control of Interference Revisited. *Theor. Comput. Sci.* 228, 1-2 (1999), 211–252. `https://doi.org/10.1016/S0304-3975(98)00359-4`

References IV

Tomas Petricek. 2018. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming* (2018).

Mark Priestley. 2011. *The Algol Research Programme*. Springer London, 225–252. https://doi.org/10.1007/978-1-84882-555-0_9

John C. Reynolds. 1978. Syntactic Control of Interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 39–46. https://doi.org/10.1145/512760.512766

Bjarne Stroustrup. 2007. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. 1–59. https://doi.org/10.1145/1238844.1238848

Bjarne Stroustrup. 2020. The Evil of Paradigms. (2020).

References V

Hayo Thielecke. 1997. *Categorical Structure of Continuation Passing Style*. Ph.D. Dissertation. University of Edinburgh.

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. `https://doi.org/10.1145/1926385.1926436`

Philip Wadler. 1991. Is there a use for linear logic? *ACM SIGPLAN Notices* 26, 9 (1991), 255–273.