Programme for a rational reconstruction of ownership in PLs

Guillaume Munch-Maccagnoni

Inria

SCALP meeting, Marseille, February 15th 2023

Introduction

- How to gain further understanding of ownership in programming languages via models in denotational semantics
- About my incursion in programming languages after this discovery
- Challenges (technical, methodological)

Introduction

Goals

- Present a set of research questions motivated by language design problems
- Locate this effort within an approach to research in 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
- Concurrent programming
- Optimizations (a lot more of valid program transformations)

Ownership/Uniqueness

Resource management (bytecomp/bytelink.ml (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 (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 in languages without GC
- Usage of a value respects a protocol (e.g. file, network connection)
- Interoperability between languages
- Fault tolerance (exception handling)

Ownership/Uniqueness

Elephant in the room

The Rust programming language represents a breakthrough for all these questions

- Resource-management inspired by C++11
- Type system for ownership & borrowing

(Matsakis and Klock II, 2014; Anderson et al., 2016)
Did it have to arise outside of academia? Why?

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)

Ownership/Uniqueness

Approaches in this area

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”).

What does semantics have to say about ownership?
Can this intuitive hierarchy be explained in semantics?

Rational reconstructions

Rational reconstructions

- Build an understanding via a refined, simplistic models where features stand by themselves
- Connect 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

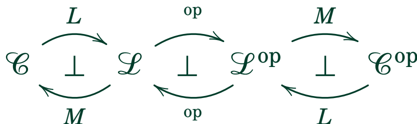
- Semantics: categorical (monad, comonad), translations (CPS, Gödel-Gentzen, into linear logic)
- Many (!) different formalisms
- Many different questions: programming (control operators), logic (classical translations)

Rational reconstructions

Example: continuations

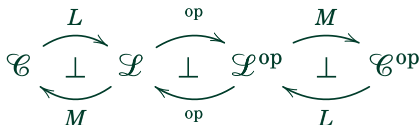
Rational reconstructions:

- 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)



Rational reconstructions

Linear call-by-push-value

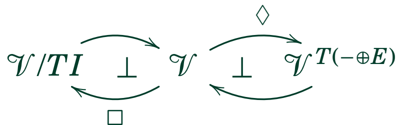


- Linear call-by-push-value (2016): how to combine *resource modalities* and *effect modalities*
- Girard: Logic of Unity (1993). Mix linear & non-linear continuations (Discussed recently: when can we duplicate continuations in OCaml?)

Rational reconstructions

A resource modality for RAII

- Combette & M. (2018). Connection between types with destructors and ordered logic.



Rational reconstructions

Co-slice category $\mathbb{O} = \mathcal{V}/TI$

- T strong monad on $\mathcal{V}$
- Objects: (A,δ) where $\delta \in \mathcal{V}(A,TI)$. (Interpretation: to any type and any effectful destruction action $\delta : A \to TI$, one associates a new type (A,δ)).
- Morphisms: $f : (A,\delta) \to (B,\delta')$ iff $f \in \mathcal{V}(A,B)$ and $\delta = \delta' \circ f$.
- Algebras of the comonad $- \times TI$ on $\mathcal{V}$.
- Fact: Any monoid structure on $M \in \mathcal{V}$ induces a monoidal structure on $\mathbb{O} = \mathcal{V}/M$.

$$TI \otimes TI \to TI$$
$$I \to TI$$

Concretely, $(A,\delta) \otimes (B,\delta') = (A \otimes B, \lambda(x,y).\delta\, x; \delta'\, y)$.

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 the inspiration from 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

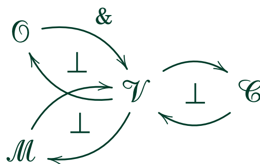
$$\mathrm{List}(A) = \mu X.(1 \oplus (A \otimes X))$$
$$\mathrm{Tsil}(A) = \mu X.(1 \oplus (X \otimes A))$$

- The stack overflow issue
  - Open problem in C++, Rust, Swift...
  - Typed pointer reversal: solution using tools 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




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

(Note: & is meant to denote borrowing (Rust), not cartesian product.)
e.g. Kind system inspired by polarities (Girard, 1991, 1993).

- *Qualitative* linearity (*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

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

- 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

Borrowing

Challenges to test the model

What is borrowing? How does it appear?

- Hypothesis: "&" (borrowing) 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

⇒ Methodological limits to the "toy system" approach

Research questions

And more

Other open problems interesting to look at from this angle

- Interrupts & failure recovery (← experience from personal contributions in the implementation in OCaml, 2018-2023)
- Limitations of Rust borrow checker (← experience from implementing an interface between Rust & OCaml, 2020-2023)

Elephant in the room

"Why did Rust have to appear outside of academia?"

Ownership in Rust =

- Heritage from C++11. *Destructors* let emerge a notion of linearity through which all other notions follow.
- Type system for ownership (from systems programming & OOP + a special ingredient: linear borrows).
  A "reboot" of C++ with better static type-checking (though not initially conceived in this way).
- and more (e.g. mixing the C++ and Erlang exception-safety models)

Elephant in the room

"Why did Rust have to appear outside of academia?"

- A LNCS book on exception-handling in the series "State-of-the-Art Survey" (Dony et al., 2006) never mentions the C++ papers on exception safety, and almost never the C++ destructors
- Visionary papers by Baker (1994; 1995) proposed a link between resources (incl. C++ destructors) and linear logic. . . yet were never cited in the functional programming community

Elephant in the room

"Why did Rust have to appear outside of academia?"

> *"It is not the ultimate purpose of a programming language to get publications in POPL—I've done one of those by mistake. The purpose is the applications I showed you. Feedback is really important, you have to talk and listen, talk and listen. Otherwise you just design the perfect solution to the wrong problem."*

Stroustrup, University of Cambridge Churchill College Annual Computer Science Lecture, 2016

Challenges in language design

- “Toy language” approach: good or bad?
  - Some phenomena appear when mixing (too) many features at once (e.g. Tov & Pucella’s affine calculus + destructors + borrowing + linear abstract types)
  - Too complex for semantics and too simple for programming practice
  - Creates experts in implementing type systems and using proof assistants (cf. drift towards type theory in PL research)
- Considerations ranging from logic to computer architecture,
  - e.g. ML: polymorphism from lambda-calculus, cache locality of minor heap.

Challenges in language design

- Requires lots of knowledge about the diverse problems faced by programmers.
  - Emergent code patterns: Is it possible to separate formal methods from language design?
  - Diminishing returns of the experience of writing compilers for the PL researcher?
  - C++ as a 40-year-long experiment?
- Difference between programs written by users, and what the language implementor imagines the programs should look like.

Challenges in language design

- Low scientific standards in some communities
  - Narrative-based papers vs. evidence-based
  - Poor bibliographical effort (e.g. laundry list of papers at the end)
  - There is more to science than making a falsifiable claim (e.g. type safety)
    Stefik & Hanenberg, *"Methodological Irregularities in Programming Language Research"* (2017)
- What standards of evidence?
  Should evidence-based mean randomized control trials (Stefik & Hanenberg)? (!)
  - *"it is perverse to criticize conferences such as ICFP for a "lack [of] empirical foundation", when the papers published there mostly do not make empirical claims"* (Gibbons, 2018) (!)

Possible solutions

Draw lessons from good examples of practical experiments

Personal favourites: Tofte et al. (2004); Tov and Pucella (2011)

- Toy languages that pretend to be a model and a mini-language at the same time (successful or not) are also non-reusable.
- Must practical experiments in language design be a multi-collaborator effort spanning many years?

Possible solutions

To allow more distance between model/toy formalism and actual language proposal, there must be a rational (not necessarily technical) discourse to connect to programming languages

- Dare talking about empirical language design criteria (lots of wisdom remains unsaid due to the fear of unsufficient technicity)
  - e.g. ML "sweet spot"
- Be sceptical of naive match between a mathematical (e.g. categorical) structure and a proposed new feature, without empirical input (see also *"the romance of mathematics"* about monads in Petricek, 2018)
- Responsibility for the "owners" of the means of production of knowledge (e.g. languages with critical mass to gather user feedback and experience)

This community is already well-armed on the mathematical side (and shows that it can have high standards).

Possible solutions

Try a way to study a programming language in parts. For instance, analyse programming languages in 3 layers:

1: Static assurances

2: Language abstractions

3: Computational behaviour

- Suggestion: find a way to focus on one aspect, without losing sight of others
- Be sceptical of works that cherry-pick without a plan to address others aspects (e.g. assuming the existence of a magical compiler that does all the algorithmic thinking)

Possible solutions

Pay attention to emerging code patterns

- Pay attention to non-academic languages & non-academic uses of programming languages? (caveats)
- Consider programs written by users as a source of knowledge about programming problems (how awful you find the code is beyond the point)

Avoid creating a religion around a set of design choices.

- State, exceptions... not written in stone
- Be wary of biases and intrinsic conflicts of interest in language design and evolution
  Stefik & Hannenberg, *"The Programming Language Wars: Questions and Responsibilities for the Programming Language Community"*, (2014)

Possible solutions

What role for Curry-Howard?

- “Linear types” are one idea that I believe receives more attention than it deserves
  - like trying to invent the notion of side effects by staring at the signature of monads

  Note: I use the terminology “linear values” (linearity of computation also exists, see strictness or Linear Haskell)

Go back at our roots

- *Structured programming* (e.g. Dijkstra):
  - Correctness should follow from the structure of the program
  - The structures provided by the programming language should facilitate reasoning about the program

  Priestley, *“The Algol Research Programme”* (2011).

Possible solutions

It is now possible, more than ever, for members of this community to have access to programming languages problems without PL researcher intermediaries.

- Accept that a researcher's role is sometimes "only" to create and disseminate knowledge about what exists and not necessarily to invent new language features.
  Maybe later one will be able to make modest observations (e.g. propose that "&" should be a homomorphism)
- Do not underestimate your abilities and legitimacy
  (cf. diminishing returns of the experience of writing compilers over the decades, blind spot of formal methods regarding emergent code patterns)
- Suggestion: linear logicians, take some time to learn Rust!

A possible key?

Stroustrup's theory of natural language evolution

"C++ is built on the idea of incremental growth and the gradual replacement of older facilities with newer ones where appropriate." (Stroustrup, 2020)

1. Evolution a language gradually based on emerging needs, come together to agree on a way forward
2. In turn, new usages found for features

(Also arguably the approach of Rust.)

A possible key?

Stroustrup’s theory of natural language evolution

1) Evolution a language gradually based on emerging needs, come together to agree on a way forward

- C++11: Move semantics
- Rust: Linear borrows (cf. *INHTPAMA*)

While preserving backwards-compatibility

- Distinguish the language specification from the living language

A possible key?

Stroustrup’s theory of natural language evolution

2) In turn, new usages found for features

- C++: Smart pointers (getting rid of new and delete, of the “rule of 5”, etc.), transactions as resources
- Rust: Interior mutability, typestate with borrowing

A possible key?

Stroustrup's theory of natural language evolution

A theory of programming language design and evolution

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

A possible key?

Seek relative claims, when one cannot find evidence for absolute ones

> *"Unfortunately, data necessary to resolve "paradigm choices" is hard to come by and available data is often ambiguous, biased, or hard to translate into concrete design choices."*

(Stroustrup, 2020)

> *"I don't do language comparisons. They are hard to do well."*

Interview with Bjarne Stroustrup, 24th Feb 2020

A possible key?

C++ as a 40-year-long experiment

> *"Resource management based on constructors and destructors was among the very first features added to C to make C++ [1982]. This work was followed up by the integration of resource management and error handling (RAII) [1994], and eventually with Howard Hinnant's work on unique_ptr and move semantics for C++11 [2002]."*

Stroustrup, Sutter, and Dos Reis, *"A brief introduction to C++'s model for type- and resource-safety"* (2015)

A possible key?

C++ as a 40-year-long experiment

> *"The central point in the exception handling design was the management of resources. [...] I noticed that many resources are released in the reverse order of their acquisition. This strongly resembles the behaviour of local objects created by constructors and destroyed by destructors."*

Stroustrup, *"A history of C++: 1979–1991"* (1993)

> *"Thus, the roots of modern C++ resource management lie in the first version of C with Classes and ultimately in my earlier work on operating systems."*

A possible key?

Structured programming

*"This is 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)

A possible key?

Structured programming

The emergence of an interesting structure (by logical and categorical standards) from such programming considerations is striking.

Conclusion

Thank you

References I

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

Henry G. Baker. 1995. "Use-Once" Variables and Linear Objects - Storage Management, Reflection and Multi-Threading. *SIGPLAN Notices* 30, 1 (1995), 45–52. https://doi.org/10.1145/199818.199860

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.

Christophe Dony, Jørgen Lindskov Knudsen, Alexander B. Romanovsky, and Anand Tripathi (Eds.). 2006. *Advanced Topics in Exception Handling Techniques*. Lecture Notes in Computer Science, Vol. 4119. Springer. https://doi.org/10.1007/11818502

References III

Jeremy Gibbons. 2018. On "Methodological Irregularities in Programming-Language Research". *Computer* 51, 4 (2018), 4–7. https://doi.org/10.1109/MC.2018.2141027

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.

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.

References IV

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

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

References V

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

Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/2661136.2661156

Andreas Stefik and Stefan Hanenberg. 2017. Methodological Irregularities in Programming-Language Research. *Computer* 50, 8 (2017), 60–63. https://doi.org/10.1109/MC.2017.3001257

References VI

Bjarne Stroustrup. 1993. A History of C++: 1979–1991. In *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 271–297. https://doi.org/10.1145/154766.155375

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).

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

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

References VII

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

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.