(dynamic (programming paradigms))

;; performance and expressivity

∼ Habilitation Defense ∽

Didier Verna

July 10 2020

Robert Strandh	University of Bordeaux, France	reporter
Nicolas Neuß	FAU, Erlangen-Nürnberg, Germany	reporter
Manuel Serrano	INRIA, Sophia Antipolis, France	reporter
Marco Antoniotti	University of Milano-Bicocca, Italy	examiner
Ralf Möller	University of Lübeck, Germany	examiner
Gérard Assayag	IRCAM, Paris, France	examiner

SORBONNE UNIVERSITÉ

EPITA ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

LRDE

Context

- **Multi-Paradigm Landscape Today**
  - Mostly heterogeneous

    Historic reasons

    Bad for software evolution and m

Lisp, Jazz, Aikido

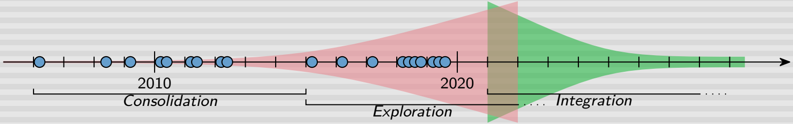
Verna, D., *The Art, Science and Engineering of Programming Journal*, 2018.

- **Special Interests**
  - Homogeneousness, dynamicity & interaction
  - OO, FP, Extensibility, reflexivity, meta-programming
- **Challenges**
  - Expressivity in homogeneous multi-paradigm environments

    Orthogonality / SoC is of the essence
  - Performance in dynamic environments

    Impact, negative or positive
- **Experimentation Platform: Lisp**
  - Subjectively: core minimalism, homoiconicity, pragmatic dialect
  - Objectively: paradigms "on steroids", official industrial standard

Timeline



1. **Consolidation**
   - Assert performance and expressivity
   - Fill in the academic bibliographic gap (*cf.* ELS)
2. **Exploration**
   - Paradigm application, extension, or design
   - Performance and/or expressivity in mind
3. **Integration**
   - Comparative assessment, paradigm junction
   - Inward / outward propagation

Outline

Introduction

Consolidation

- Performance: Optional Type Annotations
- Expressivity: Multiple Dispatch, Generic Functions, and the MOP

Exploration

Integration

Conclusion

Performance: Static *vs.* Dynamic

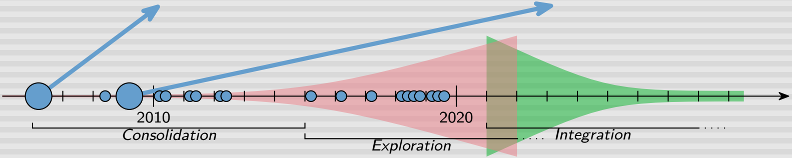
- Lisp static type annotations (weak)
- Gradual typing (Siek 2006, strong)

Beating C in Scientific Computing Applications

Verna, D., *European Lisp Workshop*, 2006.

CLOS Efficiency: Instantiation

Verna, D., *International Lisp Conference*, 2009.



Outline

Introduction

Consolidation

- Performance: Optional Type Annotations
- Expressivity: Multiple Dispatch, Generic Functions, and the MOP

Exploration

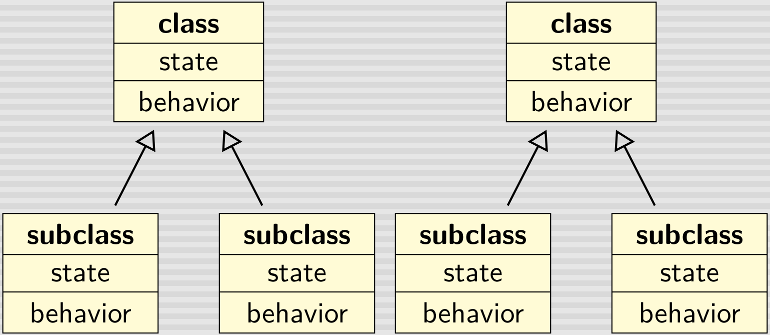
Integration

Conclusion

Expressivity: Multi-Methods and GF's

- **Multi-Methods**
  - `method (cls1 this, cls2 that, ...) {};`
- **Generic Functions**
  - Methods external to classes
  - Reified set of congruent & eponymous (multi-)methods

SoC: Multi-Methods and GF's



class
state
behavior

subclass
state
behavior

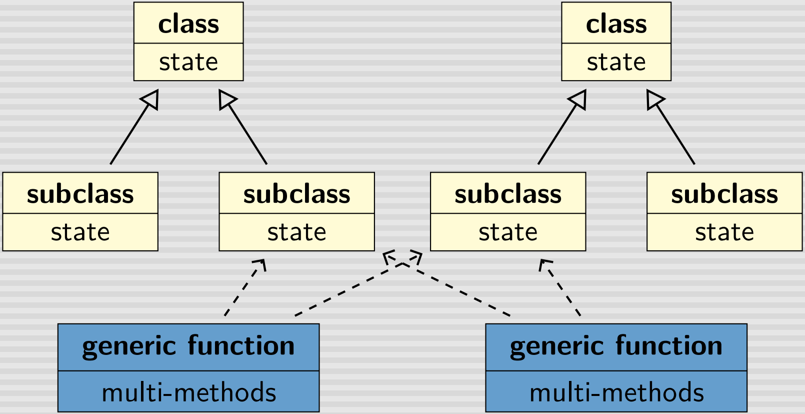
subclass
state
behavior

class
state
behavior

subclass
state
behavior

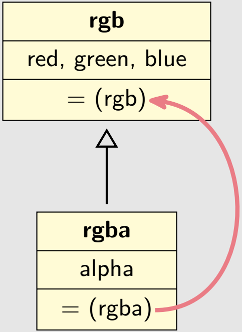
subclass
state
behavior

SoC: Multi-Methods and GF's



Binary Methods / Functions

Classical OO



rgb

red, green, blue

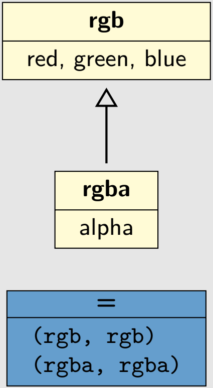
= (rgb)

rgba

alpha

= (rgba)

With generic functions



rgb

red, green, blue

rgba

alpha

=

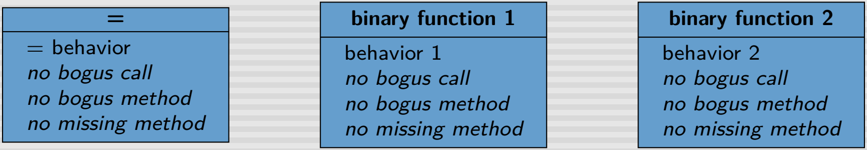
(rgb, rgb)
(rgba, rgba)

- *Cf.* Bruce (1995), Castagna (1995, 2018), *etc.*

Expressivity: the MOP

- **Safety: Protect Against Non-Conformance**
  - Binary function usage
    No bogus call (2 arguments, same class)
  - Binary function implementation
    No bogus method (2 arguments, same class)
    No missing method
- **Tools**
  - Introspection (reasonable requirement: dynamic + functional)
  - Meta-Object Protocol (cherry on the cake)

SoC: the MOP



=

= behavior
no bogus call
no bogus method
no missing method

binary function 1

behavior 1
no bogus call
no bogus method
no missing method

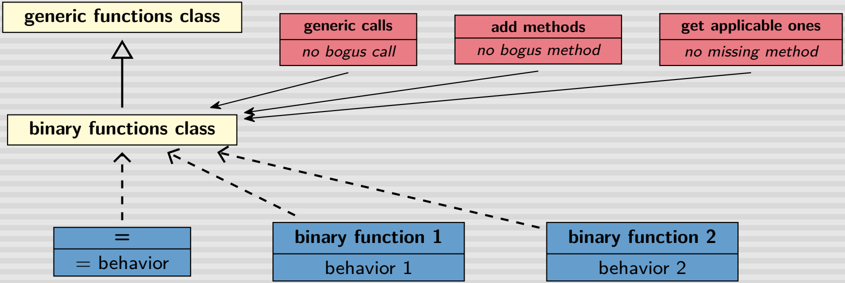
binary function 2

behavior 2
no bogus call
no bogus method
no missing method

The object layer is implemented in itself

- State = classes & instances ⇒ hierarchies extension
- Behavior = generic functions ⇒ methods specialization

SoC: the MOP


generic functions class
generic calls
no bogus call
add methods
no bogus method
get applicable ones
no missing method
binary functions class
=
= behavior
binary function 1
behavior 1
binary function 2
behavior 2

Expressivity

- Multi-paradigm & extended OO for SoC

Binary Methods Programming: the CLOS Perspective

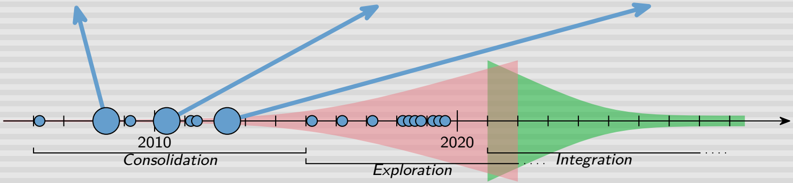
Verna, D., *Journal of Universal Computer Science*, 2008.

Revisiting the Visitor: the Just Do It Pattern

Verna, D., *Journal of Universal Computer Science*, 2010.

Extensible Languages: Blurring the Distinction between DSLs and GPLs

Verna, D., *Formal and Practical Aspects of Domain Specific Languages: Recent Developments*, chapter 1, 2012.



Outline

Introduction

Consolidation

Exploration

- Paradigm Application: Context-Oriented Optimization
- Paradigm Extension: Method Combinators
- Paradigm Design: Rational Type Expressions

Integration

Conclusion

Motivation

- **Expressivity**
  1. One thing, many different forms
  2. One form, many different things
- **Genericity: case #2**
  - *E.g.* write algorithms once, structural & behavioral details omitted
  - *Note: intensional polymorphism*
- **Problem: Genericity *vs.* Classical OO Design**
  - Object model cluttering (class / method proliferation)
  - Missed optimization opportunities (cross-cutting)
  - Performance degradation (dynamic dispatch)
- **Solution: Generative Meta-Programming**
  - ✔ Efficiency (fully-dedicated code)
  - ✘ Maintainability (w/o code)

Motivation

- Expressivity

```
template <template <class>  class M,  typename T,  typename V>
struct  ch_value_ <M <tag::value_<T>>, V>
{ typedef  M<V> ret; };

template <template <class>  class M,  typename I,  typename V>
struct  ch_value_ <M <tag::image_<I>>, V>
{ typedef  M <mln_ch_value(I, V)> ret; };

template <template <class, class>  class M,  typename T,
  typename I,  typename V>
struct  ch_value_ <M <tag::value_<T>, tag::image_<I>>, V>
{ typedef  mln_ch_value(I, V) ret; };

template <template <class, class>  class M,  typename P,
  typename T,  typename V>
struct  ch_value_ <M <tag::psite_<P>, tag::value_<T>>, V>
{ typedef  M<P, V> ret; };
```

- G itted
- P
- S

Motivation

- Expressivity

```
(BLOCK NIL
  (LET ((I 0))
    (DECLARE (TYPE (AND FIXNUM REAL) I))
    (LET ((J 1))
      (DECLARE (TYPE FIXNUM J))
      (SB-LOOP::WITH-SUM-COUNT #S(SB-LOOP::LOOP-COLLECTOR ...)
        (TAGBODY
          (WHEN (>= I '10) (GO SB-LOOP::END-LOOP))
         SB-LOOP::NEXT-LOOP
          (SETQ #:LOOP-SUM-578 (+ #:LOOP-SUM-578 (* I J)))
          (SB-LOOP::LOOP-DESETQ I (1+ I))
          (WHEN (>= I '10) (GO SB-LOOP::END-LOOP))
          (SB-LOOP::LOOP-DESETQ J (1+ J))
          (GO SB-LOOP::NEXT-LOOP)
         SB-LOOP::END-LOOP
          (RETURN-FROM NIL #:LOOP-SUM-578))))))
```

- G itted
- P
- S

Motivation

- **Expressivity**
  1. One thing, many different forms
  2. One form, many different things
- **Genericity: case #2**
  - *E.g.* write algorithms once, structural & behavioral details omitted
  - *Note: intensional polymorphism*
- **P**
  - Performance degradation (dynamic dispatch)
- **Solution: Generative Meta-Programming**
  - ✔ Efficiency (fully-dedicated code)
  - ✘ Maintainability (w/o code)

Can we be both generic and efficient within a classical OO design, without cluttering the model, and without missing cross-cutting optimization opportunities?

Context-Oriented Programming

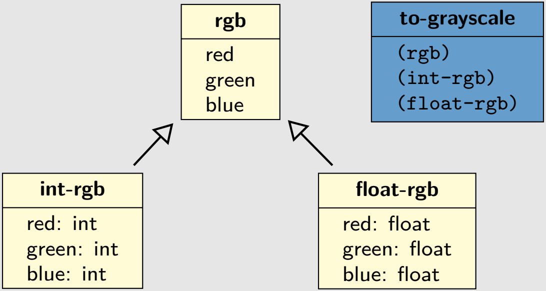
Definition

"[...] abstractions and mechanisms to concisely represent behavioral variations that depend on execution context"

- **Behavioral variation**
  - New / removed / modified behavior or structure
  - Partial definitions for software components
- **Layers**
  - Gather related context-dependent variations
  - First class citizens
- **Context**
  - Reified by sets of simultaneously active layers
  - Dynamic, late, run-time (de-)activation

Example

Static type annotations



rgb

red
green
blue

to-grayscale

(rgb)
(int-rgb)
(float-rgb)

int-rgb

red: int
green: int
blue: int

float-rgb

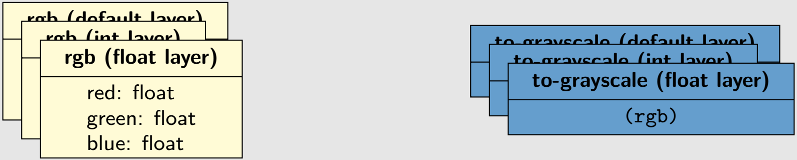
red: float
green: float
blue: float

- Dynamic types ⇒ polymorphic operations (slow)
- Subclassing ⇒ class proliferation & not cross-cutting (bad)

⟹ **Value types as contextual information**

Example

Value type layers



rgb (default layer)

rgb (int layer)

rgb (float layer)

red: float
green: float
blue: float

to-grayscale (default layer)

to-grayscale (int layer)

to-grayscale (float layer)

(rgb)

Context-Oriented Optimization

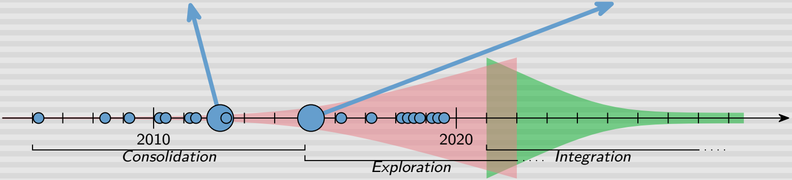
- **COP "perversion"**
  - Originally meant for pervasive and ubiquitous computing
  - Opposite of "type erasure"

Generic Image Processing with Climb

Senta, L., Chedeau, C., and Verna, D., *European Lisp Symposium*, 2012.

Context-Oriented Image Processing

Verna, D. and Ripault, F, *Context-Oriented Programming Workshop*, 2015.



Outline

Introduction

Consolidation

Exploration

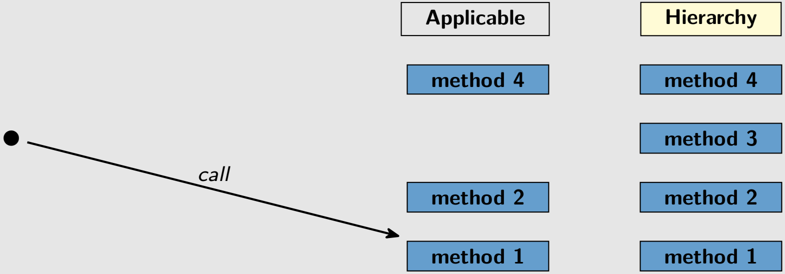
- Paradigm Application: Context-Oriented Optimization
- Paradigm Extension: Method Combinators
- Paradigm Design: Rational Type Expressions

Integration

Conclusion

Method Combinations

Classical dispatch



Applicable

Hierarchy

method 4

method 4

method 3

call

method 2

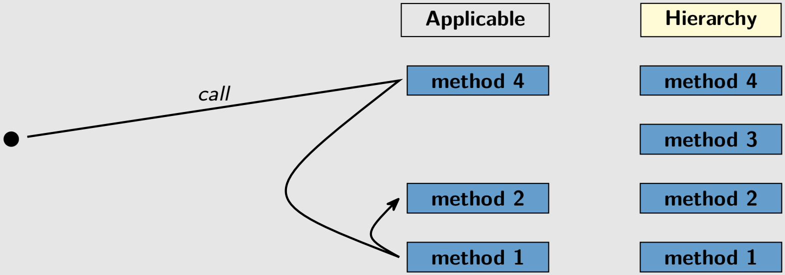
method 2

method 1

method 1

Method Combinations

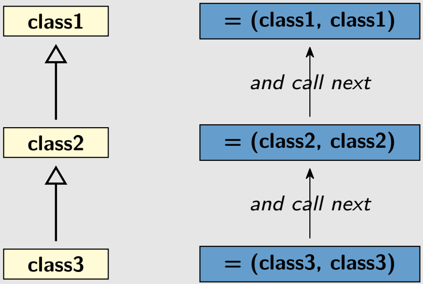
Combined dispatch



- **Built-in & programmable**
  - Method categories (qualifiers)
  - Selection & execution order
  - Participation in the final result

Example

Classical dispatch



class1

class2

class3

= (class1, class1)

and call next

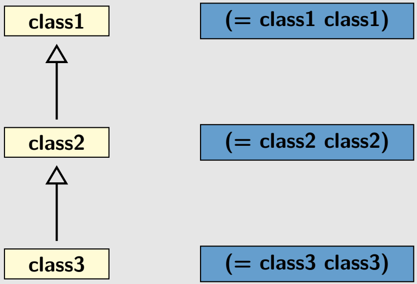
= (class2, class2)

and call next

= (class3, class3)

Example

Combined dispatch



class1

(= class1 class1)

class2

(= class2 class2)

class3

(= class3 class3)

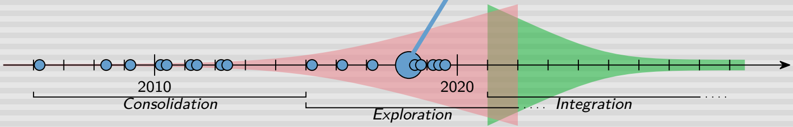
- SoC: methods code / ad-hoc dispatch code

Method Combination Problems

- **Loose Specification**
  - Lack of orthogonality (no
  - Lack of structure (no offi
  - Unclear, inconsistent or contradictory (meta-object) protocols
- **Improvement**
  - SoC: method combinations as truly global objects
  - Generic function / method combination consistency *w.r.t.* redefinitions
- **Extension: alternative combinations**

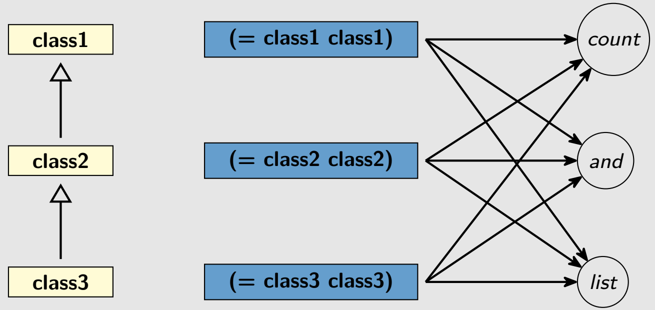
Method Combinators

Verna, D., *European Lisp Symposium*, 2018.



Example

Combined dispatch



- SoC: methods code / ad-hoc dispatch code
- Combinator SoC: generic function / method combination

Outline

Introduction

Consolidation

Exploration

- Paradigm Application: Context-Oriented Optimization
- Paradigm Extension: Method Combinators
- Paradigm Design: Rational Type Expressions

Integration

Conclusion

Heterogeneous Sequences Type Checking

- **History**
  - 2009

```
;; #### NOTE: this is where I would like a more
;; expressive dispatch in CLOS. There's in fact
;; two cases below, depending on HELP-SPEC's CAR.
(:method (sheet (help-spec list))
```

  - 2016: Jim Newton's Ph.D.
- **Problem**
  - Dynamic typing allows heterogeneous sequences
  - Optimization for homogeneous sequences
  - What about heterogeneous *yet regular* sequences?

Principle

1. **Express type regularities as rational expressions**
   - *(string . symbol* . number)* ⇐ `("foo" bar baz 42)`
2. **Provide a concrete denotation**
   - `(:. string (:* symbol) number)`
3. **Plug it into the type system**
   - `(typep value (and list (rte (:. string (:* symbol) number))))`

Example

Ad-hoc destructuring

```
(typecase elt1
  (type1 (typecase elt2
          (type1 ...)
          (type2 ...)
          ...))
  (type2 (typecase elt2
          (type1 ...)
          (type2 ...)
          ...))
  ...)
```

RTE destructuring

```
(rte-case whole-sequence
  (rte-type-1 ...)
  (rte-type-2 ...)
  ...)
```

- SoC: sequence pattern-matching code *vs.* contents processing

Rational Type Expressions

Type Checking of Heterogeneous Sequences in Common Lisp

Newton, J.E., Demaille, A., and Verna, D., *European Lisp Symposium*, 2016.

Programmatic Manipulation of Common Lisp Type Specifiers

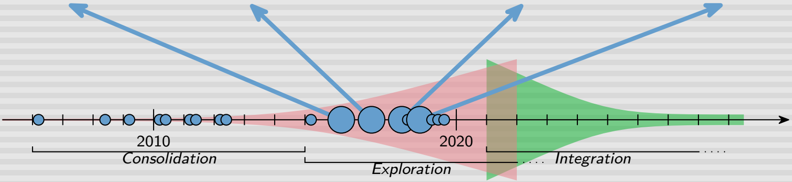
Newton, J.E., Verna, D., and Colange, M., *European Lisp Symposium*, 2017.

Strategies for Typecase Optimization

Newton, J.E., and Verna, D., *European Lisp Symposium*, 2018.

Recognizing Hetergeneous Sequences by Rational Type Expression

Newton, J.E., and Verna, D., *Workshop on Meta-Programming and Reflection*, 2018.



Outline

Introduction

Consolidation

Exploration

Integration

Conclusion

Comparative Assessment

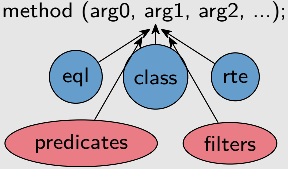
The case of Context-Oriented Optimization

- **Expressivity: Related Paradigms**
  - Aspect-Oriented Programming (Kiczales *et al.*, 1997)
  - Mixin Layers (Smaragdakis and Batory, 2001)
- **Performance: Related Solutions**
  - Generative Meta-Programming
  - JIT Compilation / Hotspot Detection

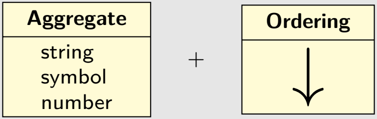
Paradigm Junction

The case of RTE

Dynamic dispatch



Nominal & structural types



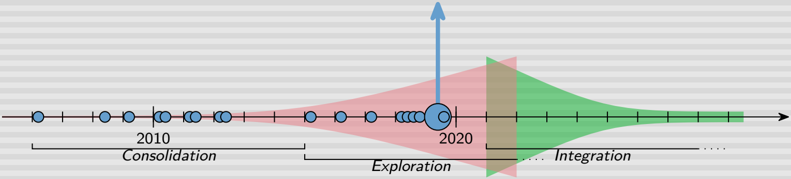
= `(rte (:. string (:* symbol) number))`

Blue Sky Territory OO Integration

- Specialization on optional / keyword arguments
- Coexistence of prototypes & classes
- Types *vs.* classes separation (*cf.* POOL, Cecil, Diesel)

Implementing Baker's SUBTYPEP Decision Procedure

Valais, L., Newton, J.E., and Verna, D., *European Lisp Symposium*, 2019.



Outline

Introduction

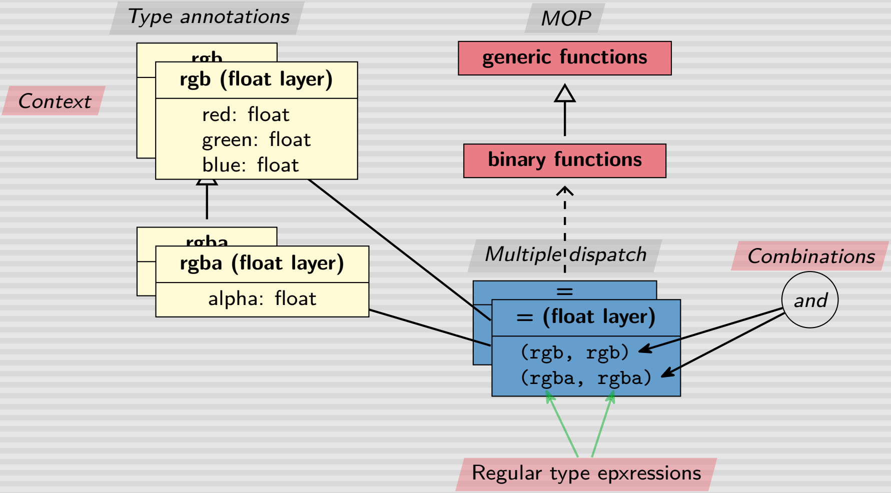
Consolidation

Exploration

Integration

Conclusion

Wrap Up



Type annotations

Context

rgb

rgb (float layer)

red: float
green: float
blue: float

rgba

rgba (float layer)

alpha: float

MOP

generic functions

binary functions

Multiple dispatch

=

= (float layer)

```
(rgb, rgb)
(rgba, rgba)
```

Combinations

and

Regular type epxressions

Conclusion

- An overview of 14 years of work (9 years full-time equivalent)
- One book chapter, 4 journal papers, 24 conference papers, >30 other presentations
- One completed Ph.D., 5 Masters internships, >10 undergraduates

Thank you!