

From Generic Invocations to Generic Implementations

PIK Potsdam, 2006/07/18.

Andreas Priesnitz and Sibylle Schupp

`{priesnit|schupp}@chalmers.se`

Department of Computing Science
Chalmers Technical University, Göteborg

Overview

- Problem: Generic Invocations yet Specific Implementations
- Goal: Generic Implementations
- Solution: Encapsulated Two-Stage Implementation Format
- Conclusions

Introduction

Generic programming relies on *invocations* (calls) and *implementations* (definitions) of *polymorphic mappings*:

- Invocation binds mapping with arguments.
- Mapping may be modeled by different algorithms.
- Algorithm may have different implementations.

Introduction

Generic programming relies on *invocations* (calls) and *implementations* (definitions) of *polymorphic mappings*:

- Invocation binds mapping with arguments.
- Mapping may be modeled by different algorithms.
- Algorithm may have different implementations.

Argument types express:

- *algorithmic semantics* (algorithm to use)
- *computational semantics* (implementation to use)

Problem Statement

Conventional generic programming methods

- lack either efficiency or simplicity, and
- express software of only restricted applicability.

Problem Statement

Conventional generic programming methods

- lack either efficiency or simplicity, and
- express software of only restricted applicability.

Common cause / symptom:

**Only invocations are generic,
but implementations are specific.**

Simplicity vs. Efficiency (Example)

Generic invocation by inefficient / invariable language or by complicated implementation techniques.

Simplicity vs. Efficiency (Example)

Generic invocation by inefficient / invariable language or by complicated implementation techniques.

Consider modeling *gcd* by *Euclidean algorithm*.

Simplicity vs. Efficiency (Example)

Generic invocation by inefficient / invariable language or by complicated implementation techniques.

Consider modeling *gcd* by *Euclidean algorithm*.

Selection of (most efficient) implementation by *computational semantics* of argument types:

- call by value (non-modifiable): `gcd(save(a), save(b))`

```
int gcd(int const& x, int const& y);
```

Simplicity vs. Efficiency (Example)

Generic invocation by inefficient / invariable language or by complicated implementation techniques.

Consider modeling *gcd* by *Euclidean algorithm*.

Selection of (most efficient) implementation by *computational semantics* of argument types:

- call by value (non-modifiable): `gcd(save(a), save(b))`

```
int gcd(int const& x, int const& y);
```

- call by reference (modifiable): `gcd(a, b)`

```
int& gcd(int& x, int& y);
```

Simplicity vs. Efficiency (Example cont.)

- static (compile-time) evaluation: `gcd(i(30),i(18))`

```
template <int X, int Y>
```

```
I<GCD<X,Y>::RESULT> gcd(I<X> x, I<Y> y);
```

Simplicity vs. Efficiency (Example cont.)

- static (compile-time) evaluation: `gcd(i(30),i(18))`

```
template <int X, int Y>  
I<GCD<X,Y>::RESULT> gcd(I<X> x, I<Y> y);
```

- (partially) lazy evaluation: `gcd(a,p(1))(b)`

```
template <...>  
Lambda<GcdTag,int&,Lambda<...> >  
gcd(int& x, Lambda<...> y);
```

Simplicity vs. Efficiency (Example cont.)

- static (compile-time) evaluation: `gcd(i(30),i(18))`

```
template <int X, int Y>  
I<GCD<X,Y>::RESULT> gcd(I<X> x, I<Y> y);
```

- (partially) lazy evaluation: `gcd(a,p(1))(b)`

```
template <...>  
Lambda<GcdTag,int&,Lambda<...> >  
gcd(int& x, Lambda<...> y);
```

- sequential / parallel evaluation

Simplicity vs. Efficiency (Example cont.)

- static (compile-time) evaluation: `gcd(i(30), i(18))`

```
template <int X, int Y>  
I<GCD<X,Y>::RESULT> gcd(I<X> x, I<Y> y);
```

- (partially) lazy evaluation: `gcd(a, p(1))(b)`

```
template <...>  
Lambda<GcdTag, int&, Lambda<...> >  
gcd(int& x, Lambda<...> y);
```

- sequential / parallel evaluation

Possibly exponential number of relevant combinations!

Simplicity vs. Efficiency

Computationally different implementations of the same algorithm lead to:

- increased development / optimization / verification / maintenance efforts
- danger of inconsistencies
- higher demands on the code developer / examiner
- repeated transfer of implementation knowledge

Simplicity vs. Efficiency

Computationally different implementations of the same algorithm lead to:

- increased development / optimization / verification / maintenance efforts
- danger of inconsistencies
- higher demands on the code developer / examiner
- repeated transfer of implementation knowledge

Common dilemma in performance-critical software!

Restricted Genericity (Example)

Multiplication of matrix and vector can be modeled by different algorithms, depending on, e.g.:

- matrix storage order (rowwise/columnwise/...)
- matrix/vector structure (dense/sparse/band/...)
- matrix/vector access (linear/random/one-time/...)

Restricted Genericity (Example)

Multiplication of matrix and vector can be modeled by different algorithms, depending on, e.g.:

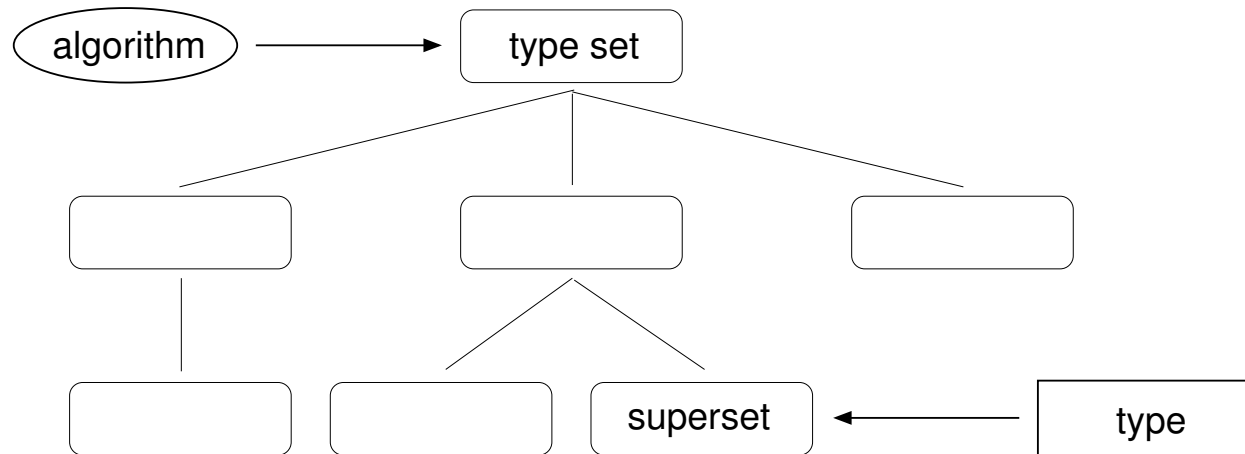
- matrix storage order (rowwise/columnwise/...)
- matrix/vector structure (dense/sparse/band/...)
- matrix/vector access (linear/random/one-time/...)

Selection of algorithm by algorithmic semantics:

- argument data and methods
- method invariants
- method performance

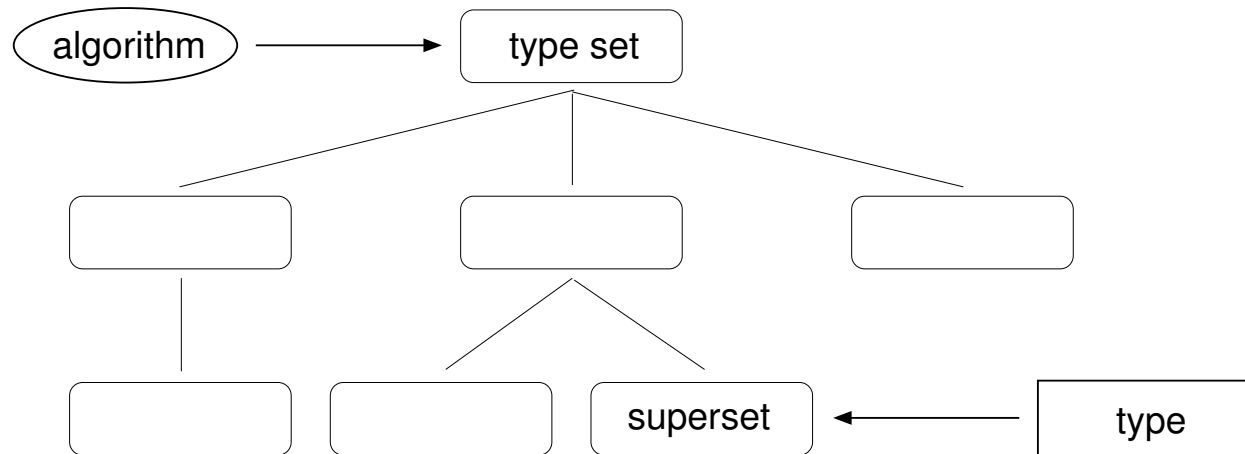
Restricted Genericity

Algorithmic argument semantics encoded as type sets
(abstract classes / concepts / type classes / ...):



Restricted Genericity

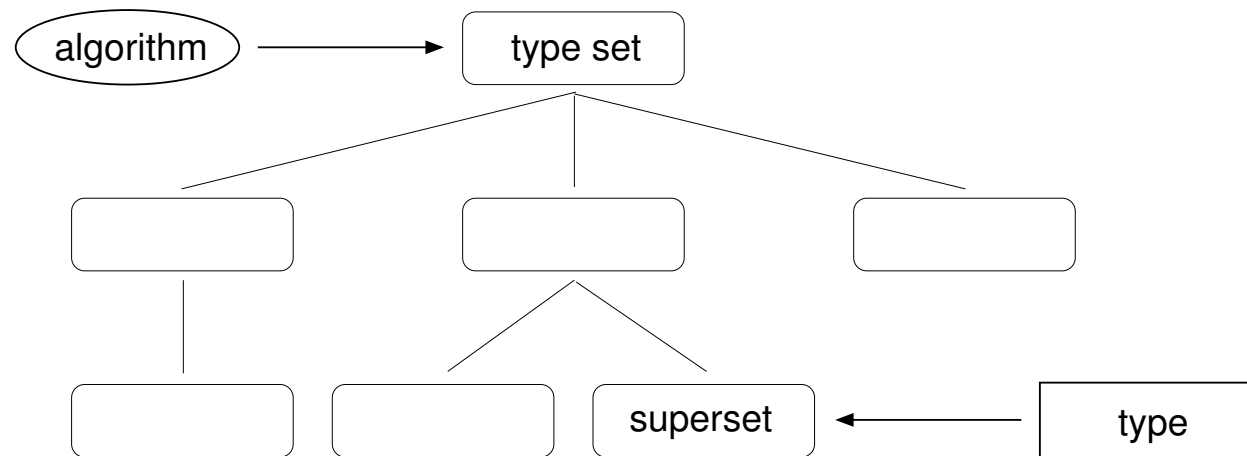
Algorithmic argument semantics encoded as type sets
(abstract classes / concepts / type classes / ...):



Many algorithms require particular argument type sets.

Restricted Genericity

Algorithmic argument semantics encoded as type sets
(abstract classes / concepts / type classes / ...):

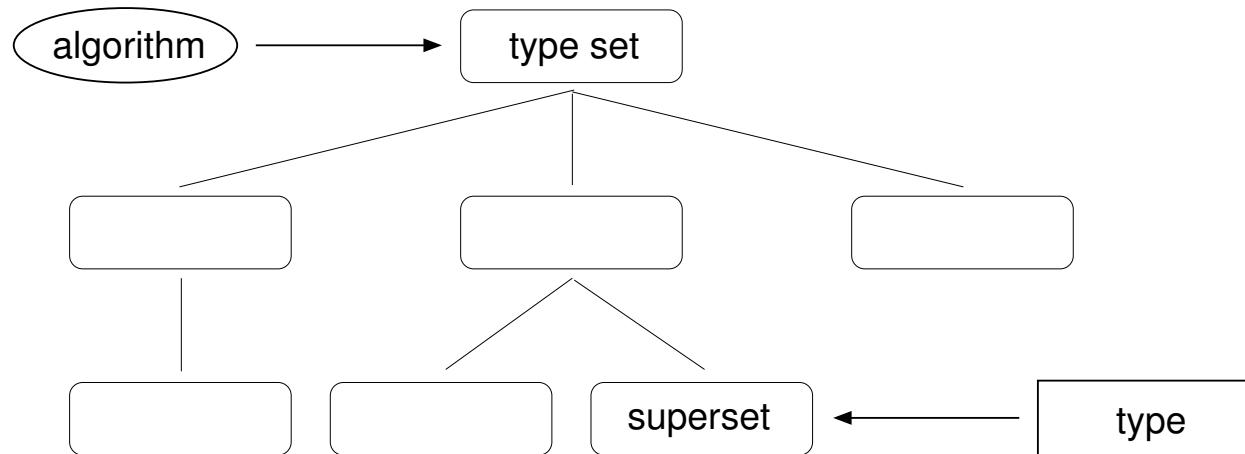


Many algorithms require particular argument type sets.

Explicit provision of all these type sets not practicable.

Restricted Genericity

Algorithmic argument semantics encoded as type sets (abstract classes / concepts / type classes / ...):



Many algorithms require particular argument type sets.
Explicit provision of all these type sets not practicable.
Common type set choices are arbitrary and restrictive.

Given some algorithm to evaluate some mapping

Given some algorithm to evaluate some mapping,
allow to devise its implementation
independent of computational variations.

Given some algorithm to evaluate some mapping,
allow to devise its implementation
independent of computational variations.

Given some mapping invocation

Given some algorithm to evaluate some mapping,
allow to devise its implementation
independent of computational variations.

Given some mapping invocation,
allow free, case-dependent selection
of the appropriate algorithm.

Generic Implementation

Find a generic mapping implementation format that:

- reflects only the algorithmic steps

$$\lambda x y . \pi_1((\textit{while} (\lambda t . \textit{not} (\textit{isNeutral} \pi_2(t))) \\ (\lambda t . \textit{tuple} \pi_2(t) (\textit{mod} \pi_1(t) \pi_2(t)))) \\ (\textit{tuple} x y))$$

Generic Implementation

Find a generic mapping implementation format that:

- reflects only the algorithmic steps

$$\lambda x y . \pi_1((\textit{while} (\lambda t . \textit{not} (\textit{isNeutral} \pi_2(t))) \\ (\lambda t . \textit{tuple} \pi_2(t) (\textit{mod} \pi_1(t) \pi_2(t)))) \\ (\textit{tuple} x y))$$

- abstracts from computational argument semantics
(state / modifiability / binding time / ...)

Generic Implementation

Find a generic mapping implementation format that:

- reflects only the algorithmic steps

$$\lambda x y . \pi_1((\textit{while} (\lambda t . \textit{not} (\textit{isNeutral} \pi_2(t))) \\ (\lambda t . \textit{tuple} \pi_2(t) (\textit{mod} \pi_1(t) \pi_2(t)))) \\ (\textit{tuple} x y))$$

- abstracts from computational argument semantics (state / modifiability / binding time / ...)
- serves as input to a function definition generator

Free Algorithm Selection

Design the implementation method to

- not restrict arguments to types or type sets

Free Algorithm Selection

Design the implementation method to

- not restrict arguments to types or type sets, but to
- provide information per potential binding

Free Algorithm Selection

Design the implementation method to

- not restrict arguments to types or type sets, but to
- provide information per potential binding, and to
- make this binding information static

Free Algorithm Selection

Design the implementation method to

- not restrict arguments to types or type sets, but to
- provide information per potential binding, and to
- make this binding information static, allowing to
- perform algorithm selection at compile-time.

Encapsulated Two-Stage Implementation

Any argument information resides in its type or value.

Encapsulated Two-Stage Implementation

Any argument information resides in its type or value.

Mapping implementations must be capable of providing any computational mapping and binding information.

Encapsulated Two-Stage Implementation

Any argument information resides in its type or value.

Mapping implementations must be capable of providing any computational mapping and binding information.

Mapping / binding information may depend

- just on the algorithm (e.g., arity)
- statically on argument types (e.g., result type)
- dynamically on argument values (e.g., result)

Encapsulated Two-Stage Implementation

Any argument information resides in its type or value.

Mapping implementations must be capable of providing any computational mapping and binding information.

Mapping / binding information may depend

- just on the algorithm (e.g., arity)
- statically on argument types (e.g., result type)
- dynamically on argument values (e.g., result)

Hence, use a (two-)staged mapping representation that encapsulates associated information.

Encapsulated Two-Stage Impl. (Example)

```
struct GcdFun
{
    // algorithm information
    template <typename Param1_, typename Param2_>
    struct Bind
    {
        // statically bound context
        typedef ... Result;
        static Result bind(Param1_ param1,
                           Param2_ param2)
        {
            // dynamically bound context
        }
    };
};
```

Specific Two-Stage Implementation

Specific two-stage implementations are provided only for *atomic* mappings and bindings

Specific Two-Stage Implementation

Specific two-stage implementations are provided only for *atomic* mappings and bindings:

- builtin operators with builtin type arguments
- traits mappings modeling type properties (e.g., conversions, qualifications, associations)
- primitive code constructs / patterns (e.g., iterations, branches, sequencing)
- legacy code (e.g., STL functions)

Specific Two-Stage Implementation

Specific two-stage implementations are provided only for *atomic* mappings and bindings:

- builtin operators with builtin type arguments
- traits mappings modeling type properties (e.g., conversions, qualifications, associations)
- primitive code constructs / patterns (e.g., iterations, branches, sequencing)
- legacy code (e.g., STL functions)

Only for implementations of these mappings / bindings all computational alternatives are manually specified!

Generic Two-Stage Implementation

By generic composition, two-stage implementations of complex mappings are automatically generated.

Generic Two-Stage Implementation

By generic composition, two-stage implementations of complex mappings are automatically generated.

All information is derived from constituting mappings.

Generic Two-Stage Implementation

By generic composition, two-stage implementations of complex mappings are automatically generated.

All information is derived from constituting mappings.

Encapsulation allows to perform composition statically and dynamically in single expression.

Generic Two-Stage Implementation

By generic composition, two-stage implementations of complex mappings are automatically generated.

All information is derived from constituting mappings.

Encapsulation allows to perform composition statically and dynamically in single expression.

Requires two-stage partial binding to represent free variables in expressions (or idiom - “lambda expression objects” in C++).

Generic Two-Stage Implementation (Ex.)

```
struct Define<GcdTag>
: As<Get<I<1>, Bind<While<BoolNot<IsNeutral
                                <Get<I<2>, P<1> > > >,
                                Tuple<Get<I<2>, P<1> >,
                                Mod<Get<I<1>, P<1> >,
                                Get<I<2>, P<1> > > > >,
                                Tuple<P<1>, P<2> > > >,
                                ...>
{};
```

Generic Two-Stage Implementation (Ex.)

```
struct Define<GcdTag>
: As<Get<I<1>, Bind<While<BoolNot<IsNeutral
                                <Get<I<2>, P<1> > > >,
                                Tuple<Get<I<2>, P<1> >,
                                Mod<Get<I<1>, P<1> >,
                                Get<I<2>, P<1> > > > >,
                                Tuple<P<1>, P<2> > > >,
                                ...>
  {};
```

$$\lambda x y . \pi_1((\textit{while} (\lambda t . \textit{not} (\textit{isNeutral} \pi_2(t)))$$
$$(\lambda t . \textit{tuple} \pi_2(t) (\textit{mod} \pi_1(t) \pi_2(t))))$$
$$(\textit{tuple} x y))$$

Special. Generic Two-Stage Implementation

User-defined types may provide special algorithm(s):

```
class HashedGCDNatural
    : public Natural
{
public:
    template <>
    struct Define<GcdTag, POSITION>
        : As< ... >
        {};
};
```

Specification depends on argument position.

Generation of User Frontend

Lexical declaration and categorization

```
DECLARE (Gcd, gcd, 2, SPECIAL, DYNAMIC, INSTANT)
```

Generation of User Frontend

Lexical declaration and categorization

```
DECLARE (Gcd, gcd, 2, SPECIAL, DYNAMIC, INSTANT)
```

generates lexical elements, e.g.:

- static function definition `Gcd`
- dynamic function definition `gcd`
- possibly operator overloadings

Generation of User Frontend

Lexical declaration and categorization

```
DECLARE (Gcd, gcd, 2, SPECIAL, DYNAMIC, INSTANT)
```

generates lexical elements, e.g.:

- static function definition `Gcd`
- dynamic function definition `gcd`
- possibly operator overloadings

and encapsulates lexical language particularities.

Two-Stage Evaluation

Evaluate a generic mapping invocation

`gcd(a, b)`

Two-Stage Evaluation

Evaluate a generic mapping invocation

```
gcd(a, b)
```

by generalized staged binding of types and values,

```
Bind<Gcd, A, B> :: bind(gcd, a, b)
```

Two-Stage Evaluation

Evaluate a generic mapping invocation

```
gcd(a, b)
```

by generalized staged binding of types and values,

```
Bind<Gcd, A, B> :: bind(gcd, a, b)
```

analyzing their computational semantics,

```
Dispatch<SGcd, SA, SB, Gcd,  $\tilde{A}$ ,  $\tilde{B}$ >  
:: Bind<Gcd, A, B> :: bind(gcd, a, b)
```

Two-Stage Evaluation

Evaluate a generic mapping invocation

```
gcd(a, b)
```

by generalized staged binding of types and values,

```
Bind<Gcd, A, B> :: bind(gcd, a, b)
```

analyzing their computational semantics,

```
Dispatch<SGcd, SA, SB, Gcd,  $\tilde{A}$ ,  $\tilde{B}$ >  
:: Bind<Gcd, A, B> :: bind(gcd, a, b)
```

and deciding on implementation alternatives,
considering static binding meta data

Two-Stage Evaluation

Evaluate a generic mapping invocation

```
gcd(a, b)
```

by generalized staged binding of types and values,

```
Bind<Gcd, A, B> :: bind(gcd, a, b)
```

analyzing their computational semantics,

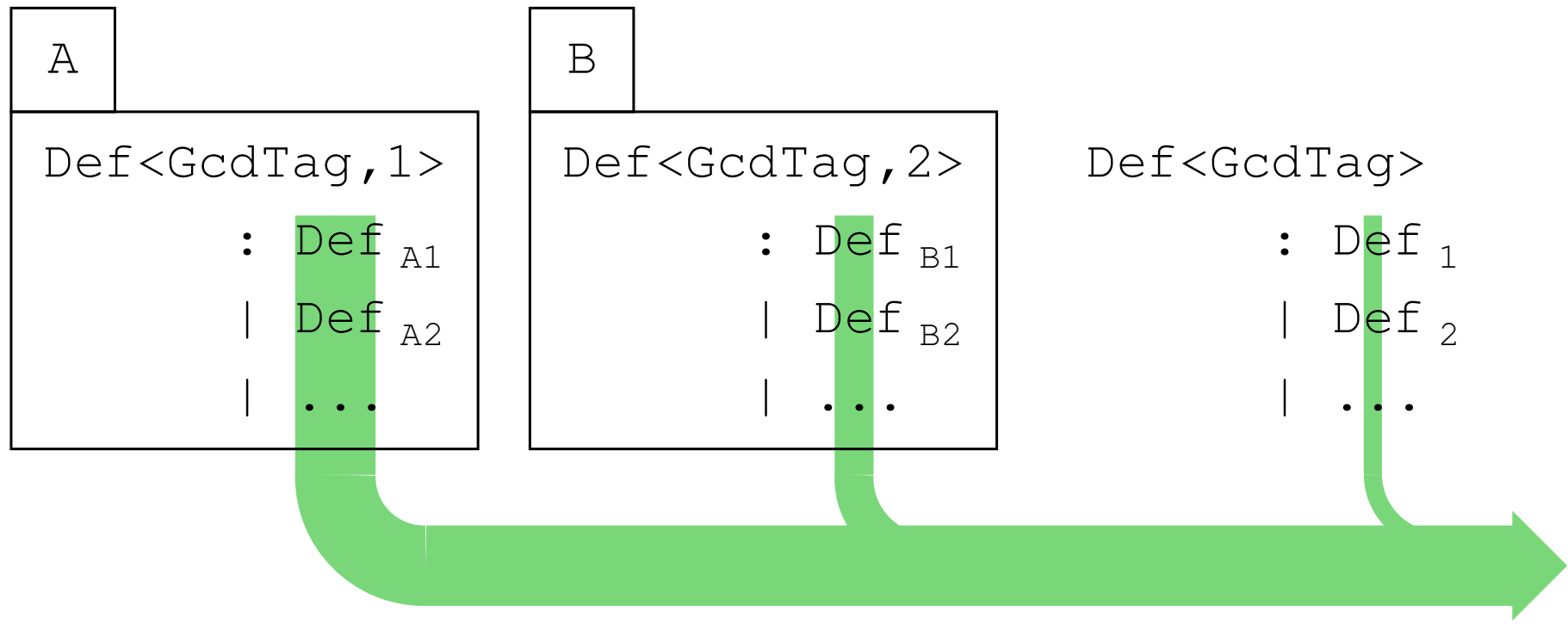
```
Dispatch<SGcd, SA, SB, Gcd,  $\tilde{A}$ ,  $\tilde{B}$ >  
:: Bind<Gcd, A, B> :: bind(gcd, a, b)
```

and deciding on implementation alternatives,
considering static binding meta data, e.g.:

binding invalid \Leftrightarrow Result is Undefined

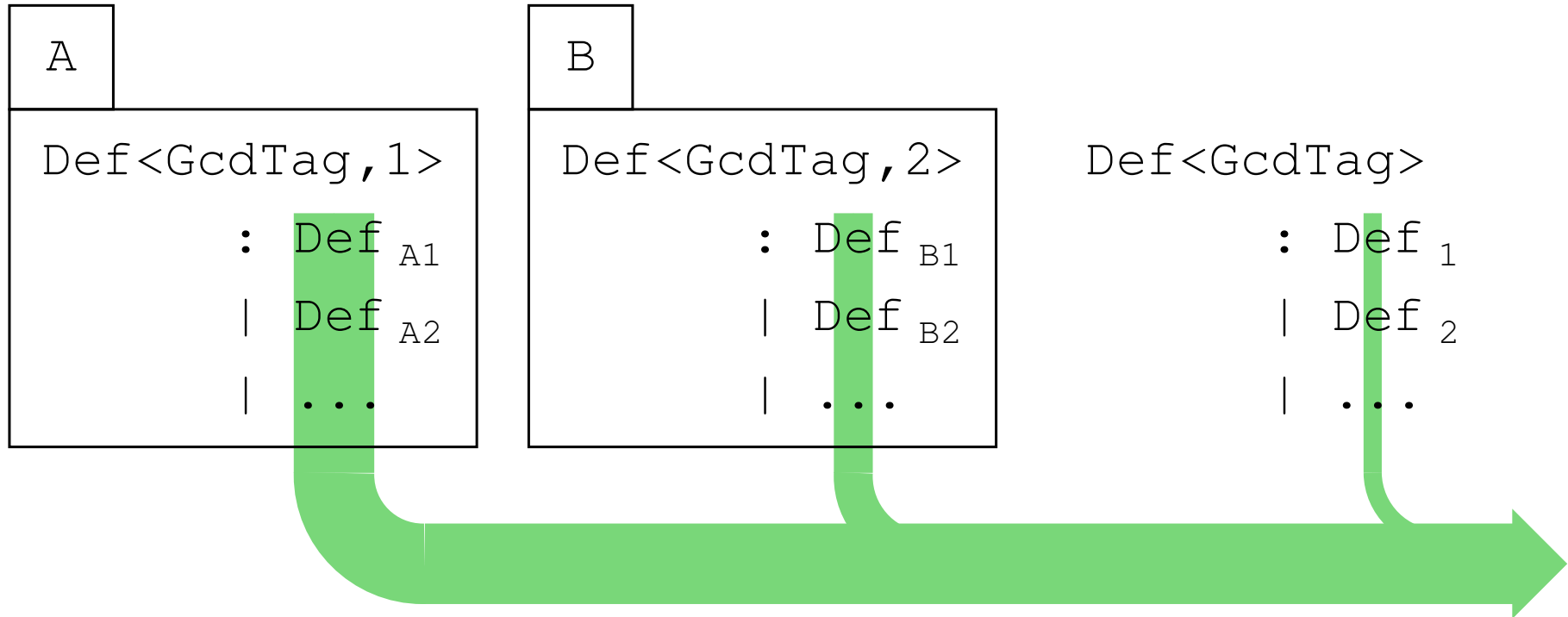
Algorithm Selection

E.g., choose first valid implementation alternative:



Algorithm Selection

E.g., choose first valid implementation alternative:



Possible extension: Provide performance data, choose most efficient (valid) alternative.

Summary of Results

- generic implementation of compositional algorithms
- encapsulation of computational issues in specific implementations of algorithmic primitives
- decoupling of algorithms and data structures
- user-defined rule for binding resolution
- uniform, simple, expressive syntax
- avoids run-time overhead
- it works!

Future Work

- identify / provide computational primitives
- increase algorithmic code base
- support implementation of data structures (algebraic / sequences / graphs / matrices / ...)
- extend supported information / methods (performance / exceptions / ...)
- formalize method and constructs
- apply to other languages?