Between abstraction and composition...

Jonathan Sterling

Logic and Semantics Seminar
Aarhus University

November 2021

Software engineering is about division of labor

Software engineering is about division of labor
between users and machines

Software engineering is about division of labor

between users and machines

between clients and servers

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Tension lies between

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Tension lies between
  abstraction (division of labor)

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Tension lies between
abstraction (division of labor)
and composition (harmony of labor).

Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Tension lies between
abstraction (division of labor)
and composition (harmony of labor).

PL theory = advancing linguistic solutions to the contradiction between abstraction and composition (paraphrasing Reynolds, 1983).

John Reynolds teaches:

> *“Type structure is a syntactic discipline for enforcing* ***levels of abstraction****.” (Reynolds, 1983)*

John Reynolds teaches:

> *"Type structure is a syntactic discipline for enforcing* ***levels of abstraction****."* *(Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

John Reynolds teaches:

> *"Type structure is a syntactic discipline for enforcing **levels of abstraction**." (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers *vs.* programmers

John Reynolds teaches:

> *"Type structure is a syntactic discipline for enforcing **levels of abstraction**." (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers	*vs.*	programmers
static	*vs.*	dynamic

John Reynolds teaches:

> *“Type structure is a syntactic discipline for enforcing* ***levels of abstraction****.” (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers	*vs.*	programmers
static	*vs.*	dynamic
public	*vs.*	private

John Reynolds teaches:

> *"Type structure is a syntactic discipline for enforcing* ***levels of abstraction****." (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers	*vs.*	programmers
static	*vs.*	dynamic
public	*vs.*	private
functions	*vs.*	algorithms

John Reynolds teaches:

> *“Type structure is a syntactic discipline for enforcing* ***levels of abstraction****.” (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers	*vs.*	programmers
static	*vs.*	dynamic
public	*vs.*	private
functions	*vs.*	algorithms
behavior	*vs.*	complexity

John Reynolds teaches:

> *"Type structure is a syntactic discipline for enforcing **levels of abstraction**." (Reynolds, 1983)*

Abstraction is a ***space***, not a binary choice.

compilers	*vs.*	programmers
static	*vs.*	dynamic
public	*vs.*	private
functions	*vs.*	algorithms
behavior	*vs.*	complexity

Thesis: need linguistic protocols to ***smoothly interpolate*** between different levels of abstraction.

We will tell our story in three parts.

1. Breaking abstraction
2. Enforcing abstraction
3. Prospects

1. Breaking abstraction

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

1. Batched queues behave like ordinary queues if you only use the queue operations.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

1. Batched queues behave like ordinary queues if you only use the queue operations.
2. Polymorphic functions list$(\alpha) \to$ list(α) can only swap, drop, and duplicate.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

1. Batched queues behave like ordinary queues if you only use the queue operations.
2. Polymorphic functions list$(\alpha) \to$ list(α) can only swap, drop, and duplicate.
3. All polymorphic functions $\alpha \to$ int are constant.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

1. Batched queues behave like ordinary queues if you only use the queue operations.
2. Polymorphic functions list(α) → list(α) can only swap, drop, and duplicate.
3. All polymorphic functions $\alpha \to$ int are constant.
4. All functions (string @ HighSecurity) → (bool @ LowSecurity) are constant.

Why do we (programmers) introduce abstraction?

To hide distracting facts *from ourselves* (and our future selves).

Protection from representation *coincidences*. An integer that encodes the name of an instruction is not to be confused with one that encodes the length of a packet.

Exploitation of representation *invariants*.

1. Batched queues behave like ordinary queues if you only use the queue operations.
2. Polymorphic functions list$(\alpha) \to$ list(α) can only swap, drop, and duplicate.
3. All polymorphic functions $\alpha \to$ int are constant.
4. All functions (string @ HighSecurity) → (bool @ LowSecurity) are constant.

Abstraction simplifies both programming and verification tasks.

Why do we break abstraction?

Why do we break abstraction?

1. Performance. To enable compiler optimizations:
    1.1 Specializing polymorphic functions (*e.g.* C++ templates)
    1.2 Exploiting unboxed representations
    1.3 Inlining code across module boundaries

Why do we break abstraction?

1. Performance. To enable compiler optimizations:
    1.1 Specializing polymorphic functions (*e.g.* C++ templates)
    1.2 Exploiting unboxed representations
    1.3 Inlining code across module boundaries
2. Debugging. Have you ever made a private member public just to debug it?

Why do we break abstraction?

1. Performance. To enable compiler optimizations:
    1.1 Specializing polymorphic functions (*e.g.* C++ templates)
    1.2 Exploiting unboxed representations
    1.3 Inlining code across module boundaries
2. Debugging. Have you ever made a private member public just to debug it?
3. **Protocol. Auctioneers may *declassify* bids after the auction is complete.**

Why do we break abstraction?

1. **Performance.** To enable compiler optimizations:
    1.1 Specializing polymorphic functions (*e.g.* C++ templates)
    1.2 Exploiting unboxed representations
    1.3 Inlining code across module boundaries
2. **Debugging.** Have you ever made a private member public just to debug it?
3. **Protocol. Auctioneers may *declassify* bids after the auction is complete.**

Most languages treat abstraction as a binary choice, but our needs are more complex.

Using *cross-module inlining* as an example, I will illustrate a path forward employing recent advances in the understanding of type theory.

Auctioneers may *declassify* bids after the auction is complete.

Modules may *declassify* bids after the auction is complete.

Modules may *declassify* private definitions after the auction is complete.

Modules may *declassify* private definitions after type checking is complete.

Separate compilation *vs.* inlining

Separate compilation = compiling each program unit as a *function* of the other units it depends on.

Separate compilation *vs.* inlining

Separate compilation = compiling each program unit as a *function* of the other units it depends on.

Pros (abstraction):

1. Enforces modularity of program units; hot-swapping?
2. Compilation can proceed in parallel

Separate compilation *vs.* inlining

Separate compilation = compiling each program unit as a *function* of the other units it depends on.

Pros (abstraction):

1. Enforces modularity of program units; hot-swapping?
2. Compilation can proceed in parallel

Cons (composition):

1. Impedes inlining and specialization of definitions
2. Impedes compiler exploitation of data representations

Separate compilation *vs.* inlining

Separate compilation = compiling each program unit as a *function* of the other units it depends on.

Pros (abstraction):

1. Enforces modularity of program units; hot-swapping?
2. Compilation can proceed in parallel

Cons (composition):

1. Impedes inlining and specialization of definitions
2. Impedes compiler exploitation of data representations

Alternative: whole-program analysis à la MLton. Works great, but very slow and memory-intensive. **We want** to put the choice in the programmer's hands.

Program units and their interfaces

Programs are divided into compilation units; units are classified by an *interface* that represents their imports and exports.

Program units and their interfaces

Programs are divided into compilation units; units are classified by an *interface* that represents their imports and exports.

Example

A fragment of the (idealized) interface to OS.FileSys in SML's Basis Library:

$$
\begin{array}{l}
\textbf{import} \\
\quad \mathsf{option} : \textbf{type} \to \textbf{type} \\
\quad \mathsf{some} : (\alpha : \textbf{type}) \to \alpha \to \mathsf{option}(\alpha), \\
\quad \mathsf{none} : (\alpha : \textbf{type}) \to \mathsf{option}(\alpha), \\
\quad \mathsf{case} : (\alpha, \beta : \textbf{type}) \to (\alpha \to \beta) \to \beta \to \mathsf{option}(\alpha) \to \beta, \\
\quad \ldots \\
\textbf{export} \\
\quad \mathsf{dirstream} : \textbf{type}, \\
\quad \mathsf{opendir} : \mathsf{string} \to \mathsf{dirstream}, \\
\quad \mathsf{readdir} : \mathsf{dirstream} \to \mathsf{option}(\mathsf{string}), \\
\quad \ldots
\end{array}
$$

Type synonyms and singleton kinds

Type synonyms = revealing the representation of a type to the programmer; *e.g.* `type dirpath = string`.

Type synonyms and singleton kinds

Type synonyms = revealing the representation of a type to the programmer; *e.g.* `type dirpath = string`.

Stone (2000) introduces *singleton kinds* $\{\mathbf{type} \hookrightarrow \tau\}$ to support revelation of representation details. An element of $\{\mathbf{type} \hookrightarrow \tau\}$ is exactly a type $\sigma : \mathbf{type}$ such that $\sigma \equiv \tau$!

Type synonyms and singleton kinds

Type synonyms = revealing the representation of a type to the programmer; *e.g.* `type dirpath = string`.

Stone (2000) introduces *singleton kinds* $\{\textbf{type} \hookrightarrow \tau\}$ to support revelation of representation details. An element of $\{\textbf{type} \hookrightarrow \tau\}$ is exactly a type $\sigma : \textbf{type}$ such that $\sigma \equiv \tau$!

$$\begin{array}{l} \textbf{export} \\ \quad \mathsf{dirpath} : \{\textbf{type} \hookrightarrow \mathsf{string}\}, \\ \quad \mathsf{dirstream} : \textbf{type}, \\ \quad \mathsf{opendir} : \mathsf{dirpath} \to \mathsf{dirstream}, \\ \quad \ldots \end{array}$$

Type synonyms and singleton kinds

Type synonyms $=$ revealing the representation of a type to the programmer; *e.g.* `type dirpath = string`.

Stone (2000) introduces *singleton kinds* $\{\textbf{type} \hookrightarrow \tau\}$ to support revelation of representation details. An element of $\{\textbf{type} \hookrightarrow \tau\}$ is exactly a type $\sigma : \textbf{type}$ such that $\sigma \equiv \tau$!

$$
\begin{array}{l}
\textbf{export} \\
\quad \text{dirpath} : \{\textbf{type} \hookrightarrow \text{string}\}, \\
\quad \text{dirstream} : \textbf{type}, \\
\quad \text{opendir} : \text{dirpath} \to \text{dirstream}, \\
\quad \ldots
\end{array}
$$

Inlining problem is similar, but we want to reveal representation details to the **compiler** but **not** the programmer. Need for *controlled abstraction breaking*.

The geometry of phase distinctions

A phase distinction is a protocol for breaking and enforcing abstraction.
Main moves: “hide information until” and “redact information from”.

Technically, phase distinctions are open/closed partitions in a space of program behaviors (*c.f.* Alpern and Schneider (1985)!).

New modal type structure to mediate between open and closed subspaces (Rijke, Shulman, and Spitters, 2020).

Let’s see how it works for our running example!

The phase distinction between compilation and programming

The inlining problem: singletons break abstraction *now*, which we want to postpone until compiletime!

The phase distinction between compilation and programming

The inlining problem: singletons break abstraction *now*, which we want to postpone until compiletime!

Let **C** be a “phase” representing compiletime.

The phase distinction between compilation and programming

The inlining problem: singletons break abstraction *now*, which we want to postpone until compiletime!

Let **C** be a “phase” representing compiletime.

- Introduce *partial singletons* $\{\tau \mid \mathbf{C} \hookrightarrow e\}$: the largest subtype of τ that becomes equivalent to the singleton $\{\tau \hookrightarrow e\}$ at compiletime.

The phase distinction between compilation and programming

The inlining problem: singletons break abstraction *now*, which we want to postpone until compiletime!

Let $\mathbf{C}$ be a "phase" representing compiletime.

- Introduce *partial singletons* $\{\tau \mid \mathbf{C} \hookrightarrow e\}$: the largest subtype of τ that becomes equivalent to the singleton $\{\tau \hookrightarrow e\}$ at compiletime.
- Phases are a partial order $\mathcal{O} = \{\mathbf{C} \leq \top\}$ where $\top$ represents "now". The (total) partial singleton $\{\tau \mid \top \hookrightarrow e\}$ is the singleton $\{\tau \hookrightarrow e\}$.

The phase distinction between compilation and programming

The inlining problem: singletons break abstraction *now*, which we want to postpone until compiletime!

Let **C** be a "phase" representing compiletime.

- Introduce *partial singletons* $\{\tau \mid \mathbf{C} \hookrightarrow e\}$: the largest subtype of τ that becomes equivalent to the singleton $\{\tau \hookrightarrow e\}$ at compiletime.
- Phases are a partial order $\mathcal{O} = \{\mathbf{C} \leq \top\}$ where $\top$ represents "now". The (total) partial singleton $\{\tau \mid \top \hookrightarrow e\}$ is the singleton $\{\tau \hookrightarrow e\}$.
- Judgments $\Gamma \vdash_{\varphi} e : \tau$ and $\Gamma \vdash_{\varphi} e \equiv e' : \tau$ are *contravariantly* indexed in phases $\varphi \in \mathcal{O}$.

Example: unboxed representations without losing abstraction

Programming-time abstractions are respected.

Example: unboxed representations without losing abstraction

Programming-time abstractions are respected. Consider a unit that imports abstract types representing the DNS protocol:

$$\begin{array}{l}
\textbf{import} \\
\quad \text{hid} : \{\textbf{type} \mid \textbf{C} \hookrightarrow \text{unsigned short}\}, \\
\quad \text{qr}, \text{opcode}, \text{aa}, \ldots : \{\textbf{type} \mid \textbf{C} \hookrightarrow \text{unsigned char}\}, \\
\quad \text{header} : \{\textbf{type} \mid \top \hookrightarrow \text{hid} \times \text{qr} \times \text{opcode} \times \text{aa} \times \ldots\}, \\
\textbf{export} \\
\quad \text{parseheader} : \text{bits} \to \text{option}(\text{header}) \times \text{bits}
\end{array}$$

Example: unboxed representations without losing abstraction

Programming-time abstractions are respected. Consider a unit that imports abstract types representing the DNS protocol:

$$\begin{array}{l}
\textbf{import} \\
\quad \mathsf{hid} : \{\textbf{type} \mid \textbf{C} \hookrightarrow \mathsf{unsigned\ short}\}, \\
\quad \mathsf{qr}, \mathsf{opcode}, \mathsf{aa}, \ldots : \{\textbf{type} \mid \textbf{C} \hookrightarrow \mathsf{unsigned\ char}\}, \\
\quad \mathsf{header} : \{\textbf{type} \mid \top \hookrightarrow \mathsf{hid} \times \mathsf{qr} \times \mathsf{opcode} \times \mathsf{aa} \times \ldots\}, \\
\textbf{export} \\
\quad \mathsf{parseheader} : \mathsf{bits} \to \mathsf{option}(\mathsf{header}) \times \mathsf{bits}
\end{array}$$

Theorem. The parser does not observably depend on the reprs of opcode, *etc.*

Example: unboxed representations without losing abstraction

Programming-time abstractions are respected. Consider a unit that imports abstract types representing the DNS protocol:

$$
\begin{array}{l}
\textbf{import} \\
\quad \mathsf{hid} : \{\textbf{type} \mid \mathbf{C} \hookrightarrow \mathsf{unsigned\ short}\}, \\
\quad \mathsf{qr}, \mathsf{opcode}, \mathsf{aa}, \ldots : \{\textbf{type} \mid \mathbf{C} \hookrightarrow \mathsf{unsigned\ char}\}, \\
\quad \mathsf{header} : \{\textbf{type} \mid \top \hookrightarrow \mathsf{hid} \times \mathsf{qr} \times \mathsf{opcode} \times \mathsf{aa} \times \ldots\}, \\
\textbf{export} \\
\quad \mathsf{parseheader} : \mathsf{bits} \to \mathsf{option}(\mathsf{header}) \times \mathsf{bits}
\end{array}
$$

Theorem. The parser does not observably depend on the reprs of opcode, *etc.*

Compilation proceeds by pulling back along the phase transition $\mathbf{C} \leq_{\mathcal{O}} \top$; we have:

$$\vdash_{\mathbf{C}} \mathsf{header} \equiv \mathsf{unsigned\ short} \times \mathsf{unsigned\ char} \times \mathsf{unsigned\ char} \times \ldots$$

⇒ unboxed repr. possible without breaking programmer-abstractions!

2. Enforcing abstraction

In the inlining example, we were hiding information *until* a phase, *e.g.* the *open modality* for phase **C**:

$$\boxed{\mathbf{C} \Rightarrow \mathsf{opcode} \equiv \mathsf{int}}$$

In the inlining example, we were hiding information *until* a phase, *e.g.* the *open modality* for phase **C**:

$$\boxed{\mathbf{C} \Rightarrow \mathsf{opcode} \equiv \mathsf{int}}$$

What about hiding information *from* a phase?

e.g. stripping and noninterference of profiling data

In the inlining example, we were hiding information *until* a phase, *e.g.* the *open modality* for phase **C**:

$$\boxed{\mathbf{C} \Rightarrow \mathsf{opcode} \equiv \mathsf{int}}$$

What about hiding information *from* a phase?

e.g. stripping and noninterference of profiling data

achieved by means of complementary *closed modality*:

$$\boxed{\mathbf{C} \vee \mathsf{int}}$$

(the smallest type containing int that becomes isomorphic to unit at compiletime)

Sealing: instrumentation *sans* interference

Poor man's profiling: add `counter` fields to some datatypes and keep track of how many times you call functions.

Sealing: instrumentation *sans* interference

Poor man's profiling: add `counter` fields to some datatypes and keep track of how many times you call functions.

Problem: your profiling code may interfere with the program behavior
$\Rightarrow$ difficult to track bugs.

Sealing: instrumentation *sans* interference

Poor man's profiling: add `counter` fields to some datatypes and keep track of how many times you call functions.

Problem: your profiling code may interfere with the program behavior
$\Rightarrow$ difficult to track bugs.

Solution: *seal* the counter variables under the closed modality $\mathbf{C} \vee \tau$; this causes them to be erased by the default compiler target.

Noninterference / modal phase splitting automatically ensures that input-output behavior of compiled programs cannot depend on the values of counters.

instrumenting a program

instrumenting a program

```
fun myfun () =

 mybody()
```

instrumenting a program

```
val counter : (C ∨ int) ref =
  ref (seal 0)

fun myfun () =


  mybody()
```

instrumenting a program

```
val counter : (C ∨ int) ref =
 ref (seal 0)

fun myfun () =
 Ref.update (Seal.map Int.incr) counter;
 mybody()
```

the program at phase **C**

```
val counter : unit ref =
 ref ()

fun myfun () =
 Ref.update (Seal.map Int.incr) counter;
 mybody()
```

the program at phase **C**

```
val counter : unit ref =
 ref ()

fun myfun () =
 Ref.update (fn () ⇒ ()) counter;
 mybody()
```

≃ *the program at phase* **C**

```
val counter : unit ref =
 ref ()

fun myfun () =
 Ref.update (fn () ⇒ ()) counter;
 mybody()
```

$\simeq$ *the program at phase* **C**

```
val counter : unit ref =
 ref ()

fun myfun () =


 mybody()
```

≃ *the program at phase* **C**

fun myfun () =

  mybody()

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g*. $\{\ell_{\text{public}} < \ell_{\text{user}} < \ell_{\text{admin}}\}$.

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g.* $\{\ell_{\mathsf{public}} < \ell_{\mathsf{user}} < \ell_{\mathsf{admin}}\}$.
2. Protocol states α are phases, *e.g.* $\{\alpha_{\mathit{done}} < \alpha_{\mathit{doing}}\}$.

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g.* $\{\ell_{\mathsf{public}} < \ell_{\mathsf{user}} < \ell_{\mathsf{admin}}\}$.
2. Protocol states α are phases, *e.g.* $\{\alpha_{\mathit{done}} < \alpha_{\mathit{doing}}\}$.
3. Closed modality $\ell_{\mathsf{user}} \vee \mathsf{string}$ hides a key from both users and the public.

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g.* $\{\ell_{\text{public}} < \ell_{\text{user}} < \ell_{\text{admin}}\}$.
2. Protocol states α are phases, *e.g.* $\{\alpha_{done} < \alpha_{doing}\}$.
3. Closed modality $\ell_{\text{user}} \vee \text{string}$ hides a key from both users and the public.
4. Open modality $\boxed{\alpha_{done} \Rightarrow \text{results}}$ reveals auction results after completion.

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g.* $\{\ell_{\mathsf{public}} < \ell_{\mathsf{user}} < \ell_{\mathsf{admin}}\}$.
2. Protocol states α are phases, *e.g.* $\{\alpha_{done} < \alpha_{doing}\}$.
3. Closed modality $\boxed{\ell_{\mathsf{user}} \vee \mathsf{string}}$ hides a key from both users and the public.
4. Open modality $\boxed{\alpha_{done} \Rightarrow \mathsf{results}}$ reveals auction results after completion.
5. Strong noninterference is immediate. What about declassification?

Sealing for information flow control

Another example is security typing & ICF using two orthogonal phase lattices.

1. Security levels ℓ are phases, *e.g.* $\{\ell_{\text{public}} < \ell_{\text{user}} < \ell_{\text{admin}}\}$.
2. Protocol states α are phases, *e.g.* $\{\alpha_{\textit{done}} < \alpha_{\textit{doing}}\}$.
3. Closed modality $\boxed{\ell_{\text{user}} \vee \text{string}}$ hides a key from both users and the public.
4. Open modality $\boxed{\alpha_{\textit{done}} \Rightarrow \text{results}}$ reveals auction results after completion.
5. Strong noninterference is immediate. What about declassification?

Relaxing noninterference with declassification seems possible by *mixing* open / closed modalities relative to authorization policy. (Stay tuned!)

3. Prospects

A menagerie of phase distinctions

open subspace	**closed subspace**	
observable properties	safety properties	Alpern and Schneider (1985)
syntax	semantics	Sterling and Angiuli (2021)
static code	dynamic code	Sterling and Harper (2021b)
compiletime	devtime	Sterling and Harper (2021a)
functions/behavior	algorithms/cost	Niu, Sterling, Grodin, and Harper (2021)

Payoff so far: syntax/semantics phase distinction was the key to *Synthetic Tait Computability*, a new kind of logical relations method that made it tractable to prove normalization for cubical type theory and representation independence for ML modules.

A metalanguage for multi-phase modularity

There are many possible phase distinctions, all with identical formal properties. Future core languages should therefore support an arbitrary phase lattice $\mathcal{O}$.

A metalanguage for multi-phase modularity

There are many possible phase distinctions, all with identical formal properties. Future core languages should therefore support an arbitrary phase lattice $\mathcal{O}$.

Three magic weapons:

A metalanguage for multi-phase modularity

There are many possible phase distinctions, all with identical formal properties. Future core languages should therefore support an arbitrary phase lattice $\mathcal{O}$.

Three magic weapons:

- *Open modality* $\boxed{\varphi \Rightarrow \tau}$ and *partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ reveal data at phase φ.

A metalanguage for multi-phase modularity

There are many possible phase distinctions, all with identical formal properties. Future core languages should therefore support an arbitrary phase lattice $\mathcal{O}$.

Three magic weapons:

- *Open modality* $\boxed{\varphi \Rightarrow \tau}$ and *partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ reveal data at phase φ.
- *Closed modality* $\varphi \vee \tau$ hides data at phase φ.

A metalanguage for multi-phase modularity

There are many possible phase distinctions, all with identical formal properties. Future core languages should therefore support an arbitrary phase lattice $\mathcal{O}$.

Three magic weapons:

- *Open modality* $\boxed{\varphi \Rightarrow \tau}$ and *partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ reveal data at phase φ.
- *Closed modality* $\varphi \vee \tau$ hides data at phase φ.
- *Fracture theorem:* any type τ is a subtype of $\boxed{\varphi \Rightarrow \tau} \times \varphi \vee \tau$.
  A complement to the Alpern–Schneider (1985) result on safety & liveness?

Prospects and future work

Several applications of the phase distinction metalanguage already developed:

- [POPL'21] A cost-aware logical framework (Niu, Sterling, Grodin, and Harper, 2021)
- [J.ACM] Logical relations as types (Sterling and Harper, 2021b)
- [LICS'21] Normalization for cubical type theory (Sterling and Angiuli, 2021)
- Normalization for multi-modal type theory (Gratzer, 2021)

Next steps:

- Develop connection to security typing and declassification (jww. Balzer and Harper)
- Generalize to support general recursion and realistic computational effects (jww. Birkedal)

Please join me! I'm looking for new collaborations.

References I

Alpern, Bowen and Fred B. Schneider (1985). “Defining liveness”. In: *Information Processing Letters* 21.4, pp. 181–185. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90056-0.

Artin, Michael, Alexander Grothendieck, and Jean-Louis Verdier (1972). *Théorie des topos et cohomologie étale des schémas*. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, 270, 305. Berlin: Springer-Verlag.

Gratzer, Daniel (2021). *Normalization for Multimodal Type Theory*. arXiv: 2106.01414 [cs.LO].

Niu, Yue, Jonathan Sterling, Harrison Grodin, and Robert Harper (2021). *A cost-aware logical framework*. Conditionally accepted to POPL ’22. arXiv: 2107.04663 [cs.PL].

Reynolds, John C. (1983). “Types, Abstraction, and Parametric Polymorphism”. In: *Information Processing*.

Rijke, Egbert, Michael Shulman, and Bas Spitters (Jan. 2020). “Modalities in homotopy type theory”. In: *Logical Methods in Computer Science* Volume 16, Issue 1. DOI: 10.23638/LMCS-16(1:2)2020. arXiv: 1706.07526 [math.CT]. URL: https://lmcs.episciences.org/6015.

Sterling, Jonathan (2021). “First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory”. PhD thesis. Carnegie Mellon University.

Sterling, Jonathan and Carlo Angiuli (July 2021). “Normalization for Cubical Type Theory”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–15. DOI: 10.1109/LICS52264.2021.9470719. arXiv: 2101.11479 [cs.LO].

Sterling, Jonathan and Robert Harper (Aug. 26, 2021a). *A metalanguage for multi-phase modularity*. ML 2021 abstract and talk. URL: https://icfp21.sigplan.org/details/mlfamilyworkshop-2021-papers/5/A-metalanguage-for-multi-phase-modularity.

— (Oct. 2021b). “Logical Relations as Types: Proof-Relevant Parametricity for Program Modules”. In: *Journal of the ACM* 68.6. ISSN: 0004-5411. DOI: 10.1145/3474834. arXiv: 2010.08599 [cs.PL].

References II

Stone, Christopher Allen (Aug. 2, 2000). “Singleton Kinds and Singleton Types”. PhD thesis. Carnegie Mellon University.