Abstraction, composition, and the phase distinction

Jonathan Sterling   j.w.w. Robert Harper
Carnegie Mellon University

August 2021

[Announcements / important dates]

- **Thesis defense** (*First Steps in Synthetic Tait Computability*) scheduled for September 13, 1:30PM. Please attend (remotely)!
- Starting postdoc at Aarhus University with Lars Birkedal in September.

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 (Reynolds, 1983).

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 (Reynolds, 1983).

This talk: separate compilation *vs.* inlining

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. Compilation can proceed in parallel
2. Enforces modularity of program units

Separate compilation *vs.* inlining

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

Pros (abstraction):

1. Compilation can proceed in parallel
2. Enforces modularity of program units

Cons (composition):

1. Prevents inlining of definitions
2. Prevents 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. Compilation can proceed in parallel
2. Enforces modularity of program units

Cons (composition):

1. Prevents inlining of definitions
2. Prevents compiler exploitation of data representations

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

Background: program units and their interfaces

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

Background: 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:

import
option : **type** $\to$ **type**
some : $(\alpha :$ **type**$) \to \alpha \to \text{option}(\alpha)$,
none : $(\alpha :$ **type**$) \to \text{option}(\alpha)$,
case : $(\alpha, \beta :$ **type**$) \to (\alpha \to \beta) \to \beta \to \text{option}(\alpha) \to \beta$,
...
export
dirstream : **type**,
opendir : string $\to$ dirstream,
readdir : dirstream $\to$ option(string),
...

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* $\{\textbf{type} \hookrightarrow \tau\}$ to support revelation of representation details. An element of $\{\textbf{type} \hookrightarrow \tau\}$ is exactly a type $\sigma : \textbf{type}$ such that $\sigma \equiv \tau$!

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 $\sigma \equiv \tau$!

export
dirpath : $\{\mathbf{type} \hookrightarrow \text{string}\}$,
dirstream : **type**,
opendir : dirpath $\rightarrow$ dirstream,
. . .

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 $\sigma \equiv \tau$!

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

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

Example: abstract types and the need for inlining

The OS.FileSys unit is generic in *any* implementation of the 'option' data type and its pattern matching principle: *algebraic data types are abstract data types* (Harper, 2013).

import
option : **type** $\to$ **type**,
some : $(\alpha$: **type**$) \to \alpha \to$ option(α),
none : $(\alpha$: **type**$) \to$ option(α),
case : $(\alpha, \beta$: **type**$) \to (\alpha \to \beta) \to \beta \to$ option$(\alpha) \to \beta$,
$\ldots$
export
dirstream : **type**,
opendir : string $\to$ dirstream,
readdir : dirstream $\to$ option(string),
$\ldots$

Example: abstract types and the need for inlining

The OS.FileSys unit is generic in *any* implementation of the 'option' data type and its pattern matching principle: *algebraic data types are abstract data types* (Harper, 2013).

import

$\mathsf{option} : \mathbf{type} \to \mathbf{type}$,
$\mathsf{some} : (\alpha : \mathbf{type}) \to \alpha \to \mathsf{option}(\alpha)$,
$\mathsf{none} : (\alpha : \mathbf{type}) \to \mathsf{option}(\alpha)$,
$\mathsf{case} : (\alpha, \beta : \mathbf{type}) \to (\alpha \to \beta) \to \beta \to \mathsf{option}(\alpha) \to \beta$,
$\ldots$

export

$\mathsf{dirstream} : \mathbf{type}$,
$\mathsf{opendir} : \mathsf{string} \to \mathsf{dirstream}$,
$\mathsf{readdir} : \mathsf{dirstream} \to \mathsf{option}(\mathsf{string})$,
$\ldots$

Good for modularity, but terrible for pattern compilation. Inlining the actual "fast path" representation is necessary!

Exporting definitions is not enough!

What about inlining of *values*? Both Stone (2000) and Leroy (2000) propose to address the inlining problem for both types and runtime values by adding singleton *types*:

$$\textbf{export } a : \{\textbf{type} \hookrightarrow \mathsf{int}\}, x : \{a \hookrightarrow 5\}$$

Exporting definitions is not enough!

What about inlining of *values*? Both Stone (2000) and Leroy (2000) propose to address the inlining problem for both types and runtime values by adding singleton *types*:

$$\textbf{export}\ a : \{\textbf{type} \hookrightarrow \text{int}\}, x : \{a \hookrightarrow 5\}$$

But: singletons reveal representation details to **both** programmer and compiler, defeating the purpose of introducing abstraction.

Why do we (programmers) use abstraction?

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

Why do we (programmers) use 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) use 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*:
  - A batched queue behaves like an ordinary queue if you only use the queue operations.
  - A polymorphic function list(α) $\rightarrow$ list(α) can only permute, drop, and duplicate elements from its input.
  - Any polymorphic function $\alpha \rightarrow$ int is constant.

Why do we (programmers) use 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*:
  - A batched queue behaves like an ordinary queue if you only use the queue operations.
  - A polymorphic function list(α) $\to$ list(α) can only permute, drop, and duplicate elements from its input.
  - Any polymorphic function $\alpha \to$ int is constant.

Abstraction simplifies both programming and verification tasks.

Why do compilers break abstraction?

Programmers introduce abstraction to protect themselves from representation details.
But compilers need these details to generate efficient code! Solutions so far:

Why do compilers break abstraction?

Programmers introduce abstraction to protect themselves from representation details.
But compilers need these details to generate efficient code! Solutions so far:

1. **Singletons à la Stone (2000), Leroy (2000)**: breaks the programmer's abstractions but allows inlining.

Why do compilers break abstraction?

Programmers introduce abstraction to protect themselves from representation details.
But compilers need these details to generate efficient code! Solutions so far:

1. **Singletons à la Stone (2000), Leroy (2000)**: breaks the programmer's abstractions but allows inlining.
2. **Break all abstractions during compilation after typechecking à la Mlton** (Weeks, 2006): very slow, can be used to fry eggs on your laptop. But preserves programmer-abstractions during typechecking-time!

Why do compilers break abstraction?

Programmers introduce abstraction to protect themselves from representation details. But compilers need these details to generate efficient code! Solutions so far:

1. **Singletons à la Stone (2000), Leroy (2000)**: breaks the programmer's abstractions but allows inlining.
2. **Break all abstractions during compilation after typechecking à la Mlton** (Weeks, 2006): very slow, can be used to fry eggs on your laptop. But preserves programmer-abstractions during typechecking-time!

We propose a unification of the two ideas, negotiated by a **phase distinction**.

Reynolds' Dictum and the Phase Distinction

What are types for? **Reynolds tells us:**

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

Reynolds' Dictum and the Phase Distinction

What are types for? **Reynolds tells us:**

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

Experience tells us: programmers and compilers work at different levels of abstraction. But our type systems do not cleanly account for this interaction.

Reynolds' Dictum and the Phase Distinction

What are types for? **Reynolds tells us:**

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

Experience tells us: programmers and compilers work at different levels of abstraction. But our type systems do not cleanly account for this interaction.

Our claim: the classic ML Family phase distinction provides crucial insight to implement Reynolds' Dictum.

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021).

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Nonetheless, other useful phase distinctions abound:

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Nonetheless, other useful phase distinctions abound:

1. syntax *vs.* semantics (Gratzer, 2021; Sterling and Angiuli, 2021; Sterling and Harper, 2021)

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Nonetheless, other useful phase distinctions abound:

1. syntax *vs.* semantics (Gratzer, 2021; Sterling and Angiuli, 2021; Sterling and Harper, 2021)
2. behavior *vs.* cost/complexity (Niu et al., 2021)

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Nonetheless, other useful phase distinctions abound:

1. syntax *vs.* semantics (Gratzer, 2021; Sterling and Angiuli, 2021; Sterling and Harper, 2021)
2. behavior *vs.* cost/complexity (Niu et al., 2021)
3. computation *vs.* specification (Melliès and Zeilberger, 2015)

Revisiting the static/dynamic phase distinction

Classic phase distinction = compiletime/static *vs.* runtime/dynamic.

Historically a pivotal notion in ML languages (Harper, Mitchell, and Moggi, 1990), re-investigated by Sterling and Harper (2021). But do we still need it?

1. Runtime values increasingly have static identity (*e.g.* SML '90, Haskell, Scala).
2. No obstruction to typechecking & compiling effectful+partial code in "full-spectrum" dependent type theory (see Lean 4, Idris 2).

Nonetheless, other useful phase distinctions abound:

1. syntax *vs.* semantics (Gratzer, 2021; Sterling and Angiuli, 2021; Sterling and Harper, 2021)
2. behavior *vs.* cost/complexity (Niu et al., 2021)
3. computation *vs.* specification (Melliès and Zeilberger, 2015)
4. **(this talk)** compilation *vs.* programming

The phase distinction between compilation and programming

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

The phase distinction between compilation and programming

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

Let **C** be a token representing the compiletime phase.

The phase distinction between compilation and programming

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

Let **C** be a token representing the compiletime phase.

- 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 to compiletime!

Let **C** be a token representing the compiletime phase.

- 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 to compiletime!

Let **C** be a token representing the compiletime phase.

- 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:

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

Example: unboxed representations without losing abstraction

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

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

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:

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

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

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

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

$\Rightarrow$ unboxed repr possible without breaking programmer-abstractions!

A metalanguage for multi-phase modularity

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

Four magic weapons:

A metalanguage for multi-phase modularity

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

Four magic weapons:

- *Partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ for breaking abstraction in phase φ.

A metalanguage for multi-phase modularity

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

Four magic weapons:

- *Partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ for breaking abstraction in phase φ.
- *Phase modality* $\langle\varphi\rangle\tau$ for code that can only be called from phase φ.

A metalanguage for multi-phase modularity

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

Four magic weapons:

- *Partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ for breaking abstraction in phase φ.
- *Phase modality* $\langle\varphi\rangle\tau$ for code that can only be called from phase φ.
- *Sealing modality* $[\varphi \setminus \tau]$ for erasing code from phase φ.

A metalanguage for multi-phase modularity

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

Four magic weapons:

- *Partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ for breaking abstraction in phase φ.
- *Phase modality* $\langle\varphi\rangle\tau$ for code that can only be called from phase φ.
- *Sealing modality* $[\varphi \setminus \tau]$ for erasing code from phase φ.
- *Fracture:* any type τ is a subtype of $(\langle\varphi\rangle\tau) \times [\varphi \setminus \tau]$.

A metalanguage for multi-phase modularity

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

Four magic weapons:

- *Partial singletons* $\{\tau \mid \varphi \hookrightarrow e\}$ for breaking abstraction in phase φ.
- *Phase modality* $\langle\varphi\rangle\tau$ for code that can only be called from phase φ.
- *Sealing modality* $[\varphi \setminus \tau]$ for erasing code from phase φ.
- *Fracture:* any type τ is a subtype of $(\langle\varphi\rangle\tau) \times [\varphi \setminus \tau]$.

Fully reconstructs static/dynamic phase distinction (see LRAT), but also refinement types (*e.g.* Liquid Haskell), parametricity/logical relations, security typing / IFC.

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 sealing (lax) modality $[\mathbf{C} \setminus \tau]$; this causes them to be erased by the default compiler. *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()
```

≃ *the program at phase* C

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

fun myfun () =


 mybody()
```

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

fun myfun () =

mybody()

Prospects and future work

Several applications of our phase distinction metalanguage already developed:

- [JACM] Parametricity for ML modules (Sterling and Harper, 2021)
- [LICS'21] Normalization for cubical type theory (Sterling and Angiuli, 2021)
- Normalization for multi-modal type theory (Gratzer, 2021)
- A cost-aware logical framework, proof-relevant type refinements (Niu et al., 2021)

Next steps:

- Develop connection to security typing/IFC (j.w.w. Balzer and Harper)
- Elaboration of high-level module constructs to metalanguage (j.w.w. Harper)
- Adapt for step-indexed logical relations (j.w.w. Birkedal)
- Prototype implementation in **cooltt** prover (j.w.w. Angiuli, Favonia, Mullanix)

References I

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.

Brady, Edwin (2021). *Idris 2: Quantitative Type Theory in Practice*. To appear in the proceedings of ECOOP 2021. arXiv: 2104.00480 [cs.PL].

De Moura, Leonardo and Sebastian Ullrich (2021). "The Lean 4 Theorem Prover and Programming Language (System Description)". To appear in the proceedings of the 28th International Conference on Automated Deduction.

Flatt, Matthew and Matthias Felleisen (1998). "Units: Cool Modules for HOT Languages". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 236–248. ISBN: 0-89791-987-4. DOI: 10.1145/277650.277730.

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

Harper, Robert (2013). *The Future of Standard ML*. Talk given at the ML Workshop. URL: https://www.cs.cmu.edu/~rwh/talks/mlw13.pdf.

Harper, Robert, John C. Mitchell, and Eugenio Moggi (1990). "Higher-Order Modules and the Phase Distinction". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, pp. 341–354. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96744.

Leroy, Xavier (May 2000). "A Modular Module System". In: *Journal of Functional Programming* 10.3, pp. 269–303. ISSN: 0956-7968. DOI: 10.1017/S0956796800003683.

References II

MacQueen, David, Robert Harper, and John Reppy (June 2020). "The History of Standard ML". In: *Proceedings of the ACM on Programming Languages* 4.HOPL. DOI: 10.1145/3386336.

Melliès, Paul-André and Noam Zeilberger (2015). "Functors are Type Refinement Systems". In: *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India: Association for Computing Machinery. ISBN: 978-1-4503-3300-9. URL: https://hal.inria.fr/hal-01096910.

Niu, Yue et al. (2021). *A cost-aware logical framework*. arXiv: 2107.04663 [cs.PL].

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

Rossberg, Andreas, Claudio Russo, and Derek Dreyer (2014). "F-ing modules". In: *Journal of Functional Programming* 24.5, pp. 529–607. DOI: 10.1017/S0956796814000264.

Sterling, Jonathan (2021). "First Steps in Synthetic Tait Computability". Forthcoming. PhD thesis. Carnegie Mellon University.

Sterling, Jonathan and Carlo Angiuli (2021). "Normalization for Cubical Type Theory". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. To appear. New York, NY, USA: Association for Computing Machinery. arXiv: 2101.11479 [cs.LO].

Sterling, Jonathan and Robert Harper (2021). "Logical Relations As Types: Proof-Relevant Parametricity for Program Modules". In: *Journal of the ACM*. To appear. arXiv: 2010.08599 [cs.PL].

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

Swasey, David et al. (2006). "A Separate Compilation Extension to Standard ML". In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: Association for Computing Machinery, pp. 32–42. ISBN: 1-59593-483-9. DOI: 10.1145/1159876.1159883.

References III

Weeks, Stephen (2006). "Whole-Program Compilation in MLton". In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: Association for Computing Machinery, p. 1. ISBN: 1-59593-483-9. DOI: 10.1145/1159876.1159877.