A metalanguage for multi-phase modularity

Jonathan Sterling Robert Harper
Carnegie Mellon University

August 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 (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.

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.

Example

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

import
option : **type** $\rightarrow$ **type**
some : $(\alpha$: **type**$) \rightarrow \alpha \rightarrow$ option(α),
none : $(\alpha$: **type**$) \rightarrow$ option(α),
case : $(\alpha, \beta$: **type**$) \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \rightarrow$ option$(\alpha) \rightarrow \beta$,
...
export
dirstream : **type**,
opendir : string $\rightarrow$ dirstream,
readdir : dirstream $\rightarrow$ option(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* $\{K \hookrightarrow \tau\}$ to support revelation of representation details. An element of $\{K \hookrightarrow c\}$ is exactly an element $c' : K$ such that $c \equiv c' : K$!

export
dirpath : {**type** $\hookrightarrow$ string},
dirstream : **type**,
opendir : dirpath $\to$ dirstream,
...

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$(\alpha) \to \beta$,
...
export
dirstream : **type**,
opendir : string $\to$ dirstream,
readdir : dirstream $\to$ option(string),
...

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

Singleton (kinds/types) are not enough

If we export definitions transparently (*e.g.* using singletons), inlining of type representations can occur.

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

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

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).

- 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 $\mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$ can only permute, drop, and duplicate elements from its input.
  - Any polymorphic function $\alpha \to \mathsf{int}$ 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:

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 MIton** (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)*

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.

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!

Let $\mathbf{C}$ 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. Consider a unit that imports abstract types representing the DNS protocol:

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

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

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

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

$\Longrightarrow$ 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 $\mathcal{O}$.

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 all aspects of existing phase distinctions, but also refinement types (*e.g.* Liquid Haskell), parametricity/logical relations, security typing / IFC.

Another example: a phase distinction for debugging

Let's be honest, `printf`-debugging is all we've got (for better or for worse); but it kills/is killed by abstraction.

Let $\mathbf{D} \in \mathcal{O}$ represent a "debugging phase", and add the following primitive:

$$\mathsf{debug} : (\langle \mathbf{D} \rangle \mathsf{string}) \to \mathsf{unit}$$

Theorem. For any $e, e' : \langle \mathbf{D} \rangle \mathsf{string}$, we have $\mathsf{debug}(e) \simeq_{obs} \mathsf{debug}(e')$.

From an abstract type, we may export a debugging function that can only be run in the debug phase:

$$\mathsf{dumpheader} : \langle \mathbf{D} \rangle (\mathsf{header} \to \mathsf{string})$$

Compilation can strip away debug code by reindexing along $(\mathbf{C} \wedge \neg \mathbf{D}) \leq_{\mathcal{O}} \top$.

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)