Typing with Continuations for the Hack Programming Language

Andrew Kennedy

Hack team, Facebook London

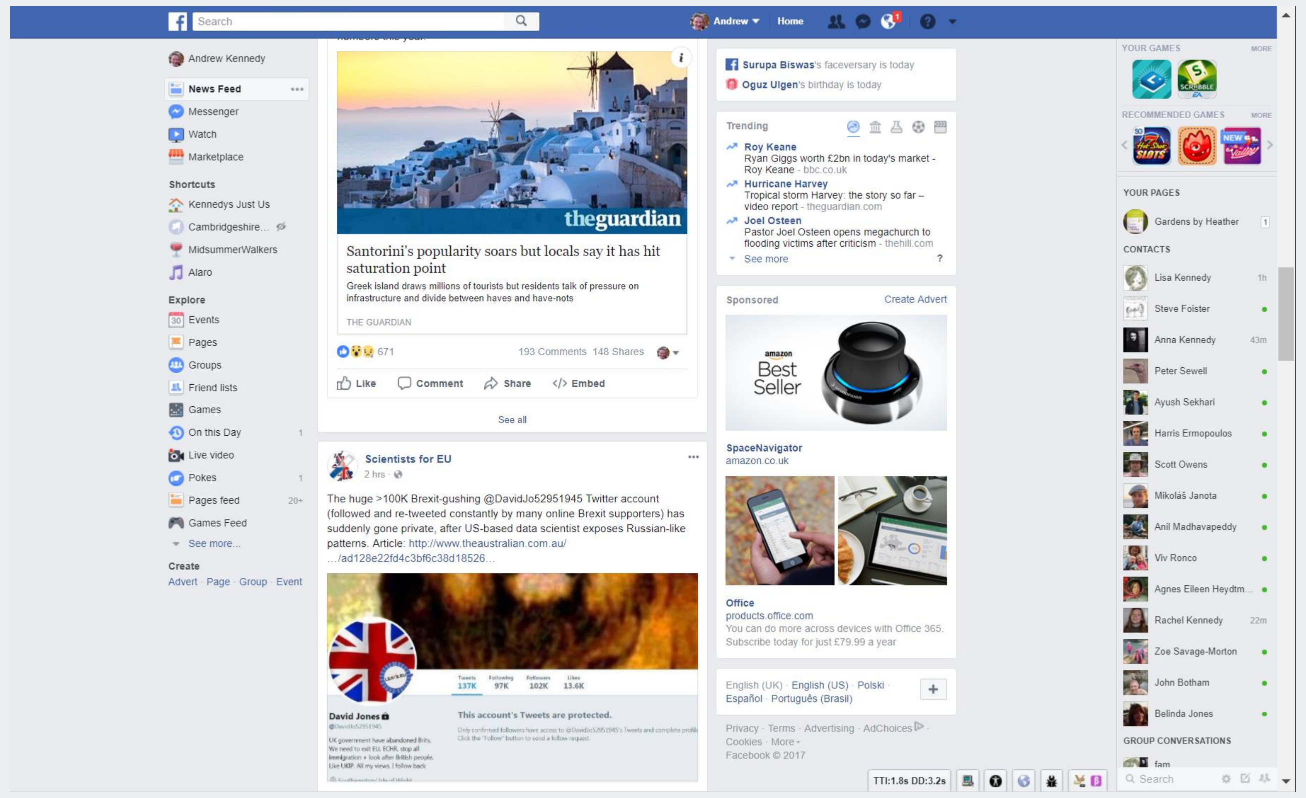
Hack: what's that?

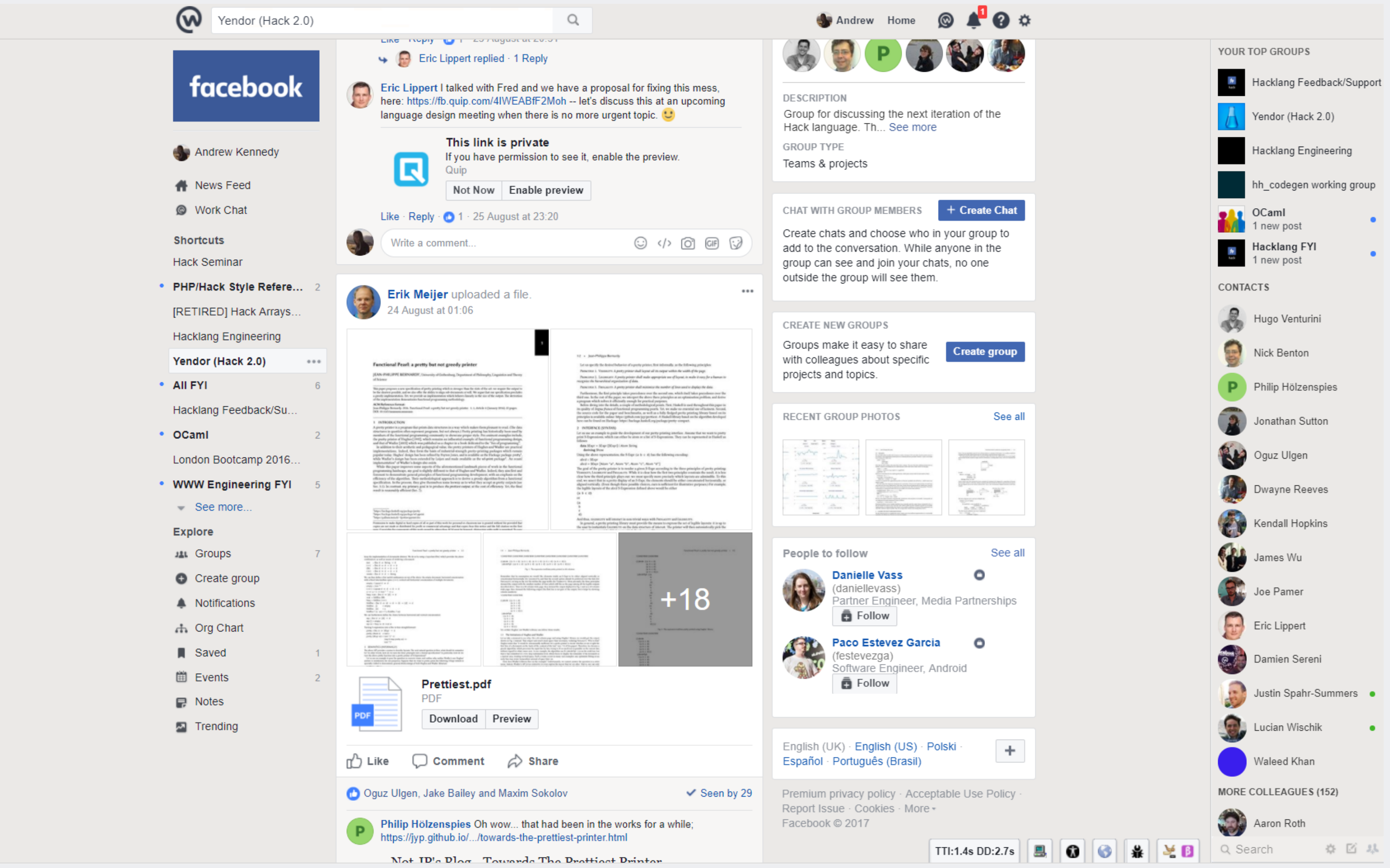
- It's Facebook's replacement for (or evolution of) PHP
  - It runs on HHVM (bytecode-based, JIT-compiled runtime)
  - Programs are checked by Hack's "whole-program" type-checker (incremental, parallel, implemented in OCaml)
- Millions of lines of PHP have been migrated to Hack, adding static types, async, and other features

Hack at Facebook

Used for “front-end” code, e.g.

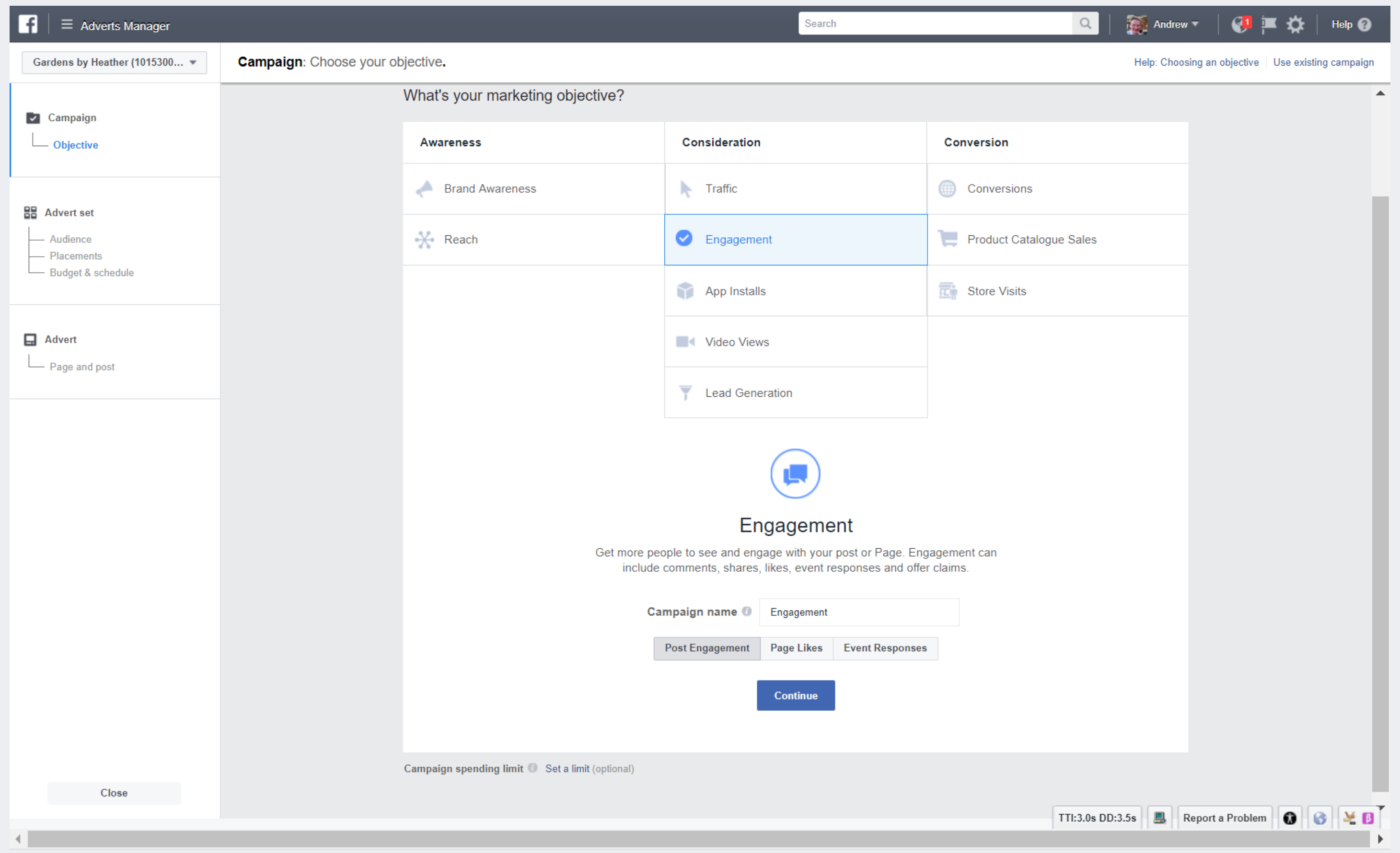
facebook.com





Workplace

Ads platform



Internal tools, etc

So what was so bad about PHP?

- Where to start…
  - == not even transitive
  - "1ne" + "2wo" evaluates to 3
  - Array access returns null for out-of-bounds
  - $a["23"] has same semantics as $a[23]
  - Stock answer to many "why?" questions we get from developers:

Because PHP!

Types in Hack

- Hack puts static types on PHP code, borrowing ideas from Java, C#, Scala:
  - OO-style subtyping (classes, interfaces, traits)
  - Non-null by default, explicit nullable ?t
  - Generics, with variance, lower/upper bounds
  - Structural subtyping: function types, shapes, tuples, arrays
  - “this” type, abstract type members

Formalizing the type system

- Long desired...
  - Write down what we think Hack's type system should be
  - Declarative: separate *type system* from *inference algorithm*
  - Use it as basis for proposing new features
  - Use it to guide correct re-implementation of existing features
- Tooling: use OTT (Sewell et al)
  - Generates LaTeX, OCaml datatypes (used in toy implementation), Coq (not yet used!)

This talk: Local variables

- Least worst feature of PHP. Scoped to function/method:
  - No declaration; created on first assignment
  - Runtime type typically changes during execution
  - Can be unset
  - Types can be tested dynamically
- (If you have a strong stomach, search for "variable variable". These are banned.)

Static typing of locals

- Flow-sensitive
  - At join points, find upper bound of types
  - Type (and null) tests *refine* types of locals
  - GADT-style treatment of type parameters in type tests

Join points

```
function f(bool $b): mixed {
  if ($b) {
    $x = 'b';
    bar($x);
    $x = 12;
  }
  else {
    $x = 'a';
  }
  return $x;
}
```

`int | string` is a subtype of `mixed` (Hack's top type)

Internally, Hack gives `$x` the type `int | string`

Type refinement

- Goal: statically check idiomatic use of type tests

```
class List<T as I> { … }
function foo(mixed $m): string {
  if (is_int($m)) {
    …
  } else if ($m instanceof List) {
    …
  }
}
```

Hack refines type of **$m** to **int**

Hack refines type of **$m** to **List<T#1>** where **T#1** is abstract, but with upper bound **T#1 <: I**

- But not general theorem proving! (e.g. no negation, conjunction in type system)

Non-standard control flow

- For example: catch, finally, break, continue:

```
function foo(int $i): string {
  $s = true;
  do {
    if ($i < 5) break;
    $s = "hey";
    $i++;
  } while ($i < 10);
  return $s;
}
```

Hack *should* report type error here

Formalizing flow sensitivity

- Key Idea: at any program point, there are a fixed number of possible *continuations*
  - The **next** statement (usual continuation)
  - The **break** continuation (in a loop, or switch)
  - The **continue** continuation (in a loop)
  - The **catch** continuation (in a try block)
  - The **finally** continuation (in a try-finally block)

Toy subset of Hack

$$\tau ::= bool \mid int \mid mixed \mid \ \dots$$

$$e \ ::= \$x \mid e_1 op\ e_2 \mid \ \dots$$

$$s \ ::= \$x = e; \mid \{\,\} \mid \{s\ \vec{s}\} \mid if\ (e)\ s_1 else\ s_2;$$

$$\mid break; \mid continue; \mid while\ (e)\ s; \mid \dots$$

Assume a subtyping relation: $\tau_1 <: \tau_2$

Typing expressions

- Define a context for locals

$$\Gamma ::= \{\, x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \}$$

- For example

$$\Gamma = \{x \colon int, y \colon bool | string\}$$

- Define typing judgment for expressions

$$\Gamma \vdash e : \tau$$

- (In real language, expressions can make assignments to locals; let's ignore that here!)

Typing statements

- Now define a context for continuations,

$$\Delta ::= \{\, k_1 : \Gamma_1, \ldots, k_n : \Gamma_n \}$$

- For example:

$$\Delta = \{next: \{x: int\}, break: \{x: string, y: bool\}\}$$

- Then define a judgment for statements

$$\Gamma; \Delta \vdash s$$

meaning "it's safe to execute s under locals Γ and continuations Δ".

Sequencing

$$\frac{}{\Gamma; next\colon \Gamma \vdash \{\ \}}$$

$$\frac{\Gamma; \Delta[next\colon \Gamma'] \vdash s \qquad \Gamma'; \Delta \vdash \{\vec{s}\}}{\Gamma; \Delta \vdash \{s; \vec{s}\}}$$

$$\frac{\Gamma; \Delta \vdash s \qquad next \notin dom(\Delta)}{\Gamma; \Delta \vdash \{s; \vec{s}\}}$$

Unreachable: might warn or error

Assignment

$$\frac{\Gamma \vdash e : \tau}{\Gamma; next : \Gamma[x : \tau] \vdash \$x = e}$$

Conditional

$$\frac{\Gamma \vdash e : bool \qquad \Gamma; \Delta \vdash s_1 \qquad \Gamma; \Delta \vdash s_2}{\Gamma; \Delta \vdash if\ (e)\ s_1\ else\ s_2}$$

Loops

$$while(e)s \equiv while(true)\{\, if(!\, e)break; s\, \}$$

$$do\ s\ while(e) \equiv while(true)\{\, s; if(!\, e)break; \}$$

$$\frac{\Gamma; \Delta[break{:}\, \Gamma', continue{:}\, \Gamma], next{:}\, \Gamma \vdash s\ ok}{\Gamma; \Delta, next{:}\, \Gamma' \vdash while(true)s}$$

$$\frac{}{\Gamma; break{:}\, \Gamma \vdash break} \qquad \frac{}{\Gamma; continue{:}\, \Gamma \vdash continue}$$

Weakening

$$\frac{\Gamma_1; \Delta_1 \vdash s \qquad \Gamma_2 <: \Gamma_1 \qquad \Delta_2 <: \Delta_1}{\Gamma_2; \Delta_2 \vdash s}$$

$$\frac{\tau_1 <: \tau_2}{\Gamma, x : \tau_1 <: \Gamma, x : \tau_2} \qquad \frac{}{\Gamma, x : \tau <: \Gamma}$$

$$\frac{\Gamma_1 <: \Gamma_2}{\Delta, k : \Gamma_2 <: \Delta, k : \Gamma_1} \qquad \frac{}{\Delta, k : \Gamma <: \Delta}$$

Implementing flow sensitivity

- Define inference function Inf so that

  $$Inf(\Gamma, s) = \Delta$$

  produces the weakest Δ such that $\Gamma; \Delta \vdash s$ holds (cf strongest post-condition in Hoare logic).

Inference (sequencing, assignment)

$$Inf(\Gamma, \$x = e) = let\ \tau = Inf(\Gamma, e) in\ \{next: \Gamma[x: \tau]\}$$

$$Inf(\Gamma, \{\ \}) = \{next: \Gamma\}$$

$$Inf(\Gamma, \{s; \vec{s}\}) =$$

$$let\ \Delta_1 = Inf(\Gamma, s)\ in$$

$$let\ \Delta_2 = Inf(\Delta_1(next), \vec{s})\ in$$

$$(\Delta_1 \setminus next) \sqcap \Delta_2$$

Could warn or error if
this doesn't exist (unreachable code)

Inference (conditional)

$$
\begin{aligned}
&Inf(\Gamma, if\ (e) s_1\ else\ s_2) = \\
&\quad check(Inf(\Gamma, e) <: bool) \\
&\quad let\ \Delta_1 = Inf(\Gamma, s_1) in \\
&\quad let\ \Delta_2 = Inf(\Gamma, s_2) in \\
&\quad \Delta_1 \sqcap \Delta_2
\end{aligned}
$$

Inference (loop)

$$Inf(\Gamma, break) = \{break: \Gamma\}$$
$$Inf(\Gamma, continue) = \{continue: \Gamma\}$$

$$
\begin{aligned}
& Inf(\Gamma, while(true)s) = \\
& \quad let\ rec\ iter(\Gamma) = \\
& \quad\quad let\ \Delta = Inf(\Gamma, s)\ in \\
& \quad\quad\quad if\ \Delta(next) <: \Gamma \wedge \Delta(continue) <: \Gamma \\
& \quad\quad\quad\quad then\ \{next: \Delta(break)\} \\
& \quad\quad\quad else\ iter(\Gamma \sqcup \Delta(next) \sqcup \Delta(continue)) \\
& \quad in\ iter(\Gamma)
\end{aligned}
$$

Operations on contexts

$$\Delta_1 \sqcap \Delta_2 = \{\, k : \Gamma_1 \sqcup \Gamma_2 \mid \Delta_1(k) = \Gamma_1, \Delta_2(k) = \Gamma_2\} \cup$$
$$\{\, k : \Gamma \mid \Delta_1(k) = \Gamma, k \notin dom(\Delta_2)\,\} \cup$$
$$\{\, k : \Gamma \mid \Delta_2(k) = \Gamma, k \notin dom(\Delta_1)\}$$
$$\Gamma_1 \sqcup \Gamma_2 = \{\, x : \tau_1 \sqcup \tau_2 \mid x : \tau_1 \in \Gamma_1, x : \tau_2 \in \Gamma_2\}$$

Choose how to interpret $\sqcup$ on types e.g.

- Find named upper bound (e.g. mixed)
- Union types in language (this is what we do in Hack)

Type refinement

$$\frac{\Gamma \vdash \$x : \tau \quad \Gamma[x : \tau \sqcap \tau']; \Delta \vdash s_1 \quad \Gamma[\mathrm{x} : \tau \setminus \tau']; \Delta \vdash s_2}{\Gamma; \Delta \vdash if\ (\$x\ is\ \tau')\ s_1\ else\ s_2}$$

$$\begin{aligned}
&Inf(\Gamma, if\ (\$x\ is\ \tau')\ s_1\ else\ s_2) = \\
&\quad let\ \tau = \Gamma(x)\ in \\
&\quad let\ \Delta_1 = Inf(\Gamma[x : \tau \sqcap \tau'], s_1)\ in \\
&\quad let\ \Delta_2 = Inf(\Gamma[x : \tau \setminus \tau'], s_2)\ in \\
&\quad \Delta_1 \sqcap \Delta_2
\end{aligned}$$

Other features

Continuation-based approach extends nicely to

- Switch (break, drop-through)
- Try (catch, and finally continuations)
- Unset (simply drop variable from context)
- (Horrible feature: break n, via stack of break continuations)

Type refinement with existentials

```
class C<T> {
function __construct(public T $item) { }
}
function findpair(vec<mixed> $v):bool {
  $i = 0;
  $first = $second = null;
  while (true) {
    if ($i >= count($v)) return false;
    $x = $v[$i];
    if ($x instanceof C) {
      if ($first === null) {
        $first = $x; continue;
      } else {
        $second = $x; break;
      }
    }
    $i++;
  }
  $first->item = $second->item;
```

- Tricky: need scoped type parameters in the context?

Unsound: `$first` has type `C<T>` for some `T`, `$second` has type `C<T>` for some possibly-different `T`

Summary

- Messy language, elegant typing rules
- Use continuations for a variety of standard control flow constructs
- New inference algorithm is a big improvement over what we had
  - Now sound
  - Captures “unreachability” and defined-ness for free (not a separate pass)
  - Performant (despite tracking multiple continuations)