Compiling and Typing with Continuations

Andrew Kennedy
Meta London

Continuations

One of the best and oldest ideas in Computer Science: 60 years old, with many many applications

Logic

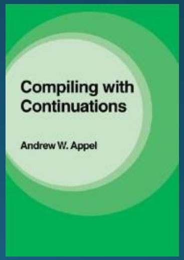
$\neg\neg\psi$

Semantics

$[\![int]\!] = (\mathbb{Z} \to R) \to R$

Distributed systems

Compilers

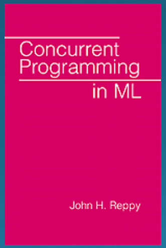


User interface modelling

Programming Language Design

Racket

Concurrency



Web programming

ocsigen

"Thinking continuations": try-catch

Q: What's wrong with OCaml's try-with construct?

A: Sometimes it's clumsy to use (see paper), but hard to put your finger on why, or what would work better.

J. Functional Programming 11 (4): 395–410, July 2001. Printed in the United Kingdom 395

Exceptional syntax

NICK BENTON and ANDREW KENNEDY
Microsoft Research, St. George House, 1 Guildhall Street, Cambridge CB2 3NH, UK
(e-mail: {nick,akenn}@microsoft.com)

Abstract

From the points of view of programming pragmatics, rewriting and operational semantics, the syntactic construct used for exception handling in ML-like programming languages, and in much theoretical work on exceptions, has subtly undesirable features. We propose and discuss a more well-behaved construct.

Solution: "think continuations". A better-behaved construct has both failure *and* success *continuations* (cf "double-barrelled CPS"). Perhaps, a generalized let:

Failure

let x = e unless E -> handler in e'

Success

Or, as introduced in OCaml ten years ago, generalized match:

match e with p_1 -> e_1 | … | p_n -> e_n | exception E -> handler

Successes

Failure

"Thinking continuations": Hoare logic

- Traditionally, Hoare "triples" have been used to reason formally about imperative programs

{P} C {Q}

Precondition

Code

Postcondition

- Rough meaning (for partial correctness): if program state satisfies P at entry to C, then it will satisfy Q at exit
- This can be broken down using more primitive notion of "safe to run from L under P". Write Safe(L,P). Then say

Safe(exit,Q) => Safe(entry, P)

"Continuation-passing"

Compiling and Typing with Continuations

Andrew Kennedy
Meta London

Compiler Intermediate Representations (IR)

Functional languages

- Desugared abstract syntax
- Core lambda-calculus-like language, no restrictions
- ANF: Administrative Normal Form (canonical form lambda-calculus)
- Monadic intermediate language
- Continuation-passing style

Close to source

↓

Far from source

Imperative languages

- Desugared abstract syntax
- Flow-graph with local variables
- SSA (Static Single Assignment)
- Gated SSA
- Continuation-passing style

Compiler Intermediate Representations (IR)

Functional languages

- Desugared abstract syntax
- Core lambda-calculus-like language, no restrictions
- ANF: Administrative Normal Form (canonical form lambda-calculus)
- Monadic intermediate language
- Continuation-passing style

Close to source

↓

Far from source

Imperative languages

- Desugared abstract syntax
- Flow-graph with local variables
- SSA (Static Single Assignment)
- Gated SSA
- Continuation-passing style

Let's start by thinking about functional languages

ANF: administrative normal form

Every intermediate computation is named. Example (using OCaml syntax):

```
let rec map f xs =
  match xs with
   | [] => []
   | x::xs' => f x :: map f xs'
```

➡

```
let rec map f xs =
  match xs with
   | [] => []
   | x::xs' =>
    let y = f x in
    let ys = map f xs' in
    y::ys
```

Monadic intermediate language is similar, except that lets can be “nested”. Monads also provide a place to “hang” effects.

Common feature: order of evaluation is made explicit.

Continuation-passing style

Every function takes an additional "continuation" function that gets passed the result on return.

```
let rec map f xs =
  match xs with
  | [] => []
  | x::xs' => f x :: map f xs'
```

"CPS-convert" →

```
let rec map f xs k =
  match xs with
  | [] => k []
  | x::xs' =>
    f x (fun y ->
    map f xs' (fun ys ->
    k (y::ys))
```

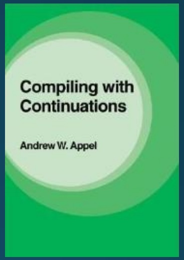
The transformation is easy to write down, but

- It "explodes" the code with a lot of new functions
- It messes with your head to think about!

As with ANF/monads, it makes order of evaluation explicit.

Why compile using CPS?

1. For languages that support *first-class control* (e.g. Scheme, Typed Racket) there is an easy CPS implementation of call-with-current-continuation (call/CC) and related features
2. Every function call becomes a tail-call => we don't need a "call stack" => all functions are closures and so instead of stack frames we have "environments" allocated on the heap. (But worry about performance!)



3. CPS-translation makes for a uniform, simple intermediate language in which we can give names to every intermediate value *and* control point. Static analyses become simpler. Some optimizations fall out really easily.

ANF is not closed under inlining

- Problem with ANF and monadic languages: they're not closed under ordinary β reduction, and terms must be "re-normalized"

let x = (λy. let z = a b in c) d in e

↓ Beta reduce (inline function)

let x = (let z = a b in c) in e — Not in ANF (no nested lets)

↓ Re-normalize

let z = a b in (let x = c in e)

Worse: conditionals

- When renormalizing conditionals in ANF or monadic form, we must duplicate the term, or introduce a lambda to represent a “join point”:

let z = (λx. If x then a else b) c in M

↓ Beta reduce (inline function)

let z = (if c then a else b) in M

Not in ANF (no let-binding of conditionals)

Re-normalize (duplicate) ↓

if c
then let z = a in M
else let z = b in M

OR: re-normalize (share) ↓

let k = λz.M in
if c
then let z = a in k z
else let z = b in k z

“Second-class” continuations

Continuations are distinguished from ordinary functions (in OCaml-like syntax):

```
let rec map f xs =
  match xs with
   | [] => []
   | x::xs’ => f x :: map f xs’
```

→

```
let rec map f xs k =
  match xs with
   | [] => k []
   | x::xs’ =>
    let cont k1(y) =
     let cont k2(ys) =
      let r = y::ys in k(r)
     in map f xs’ k2
  in f x k1
```

Local continuation definition

Continuation application

Function application

Theory and Practice

Compiling with Continuations, Continued

Andrew Kennedy
Microsoft Research Cambridge
akenn@microsoft.com

Abstract

We present a series of CPS-based intermediate languages suitable for functional language compilation, arguing that they have practical benefits over direct-style languages based on A-normal form (ANF) or monads. Inlining of functions demonstrates the benefits most clearly: in ANF-based languages, inlining involves a re-

so monads were a natural choice for separating computations from values in both terms and types. But, given the history of CPS, probably there was also a feeling that "CPS is for call/cc", something that is not a feature of Standard ML.

Recently, the author has re-implemented all stages of the SML.NET compiler pipeline to use a CPS-based intermediate lan-

ICFP'07

Jane Street's development of IR for OCaml ("Flambda 2"), is based on language described here

OCaml'23

Efficient OCaml compilation with Flambda 2

Flambda 2 team
OCamlPro
Jane Street

Abstract

Flambda 2 is an IR and optimisation pass for OCaml centred around inlining. We discuss the engineering constraints that shaped it and the overall structure that allows the compiler to be fast enough to handle very large industrial code bases.

Example 2nd-class CPS-based IR

V, W ::=
(x , y)
| $\text{in}_i\ x$
| $\lambda\ k\ x.\ M$

M, N ::=
| let $x = V$ in M
| let cont k x = M in N
| $k\ x$
| $f\ k\ x$
| let $(x,y) = z$ in M
| match x with k_1 | k_2

Explicit return

All values are named

Local continuation definition

Continuation application

All function applications take a continuation argument

Branches are named (design choice)

Features of the CPS language

- All intermediate values are named; all control points are named
  - Consequence 1: only ever substitute variables for variables
  - Consequence 2: “join points” are present from the start.
- Continuations are *second-class*: they can be passed to functions, but not returned, or stored in data structures, or accessed from outer function scopes
  - Consequence 1: local continuation definition can be implemented by a code block
  - Consequence 2: continuation application can be implemented by a jump or return
- Open question: how should continuation definitions be nested? Outermost (closed with respect to free variables)? Or innermost (minimal parameters)?

CPS *is* closed under inlining

(λy k. a b (λz k. c)) d (λx.e)

⇓ Beta reduce (argument and continuation)

a b (λz. (λx. e) c)

let f = λk x. if x then k a else k b in
let j z = M in
f j c

⇓ Beta reduce (argument and continuation)

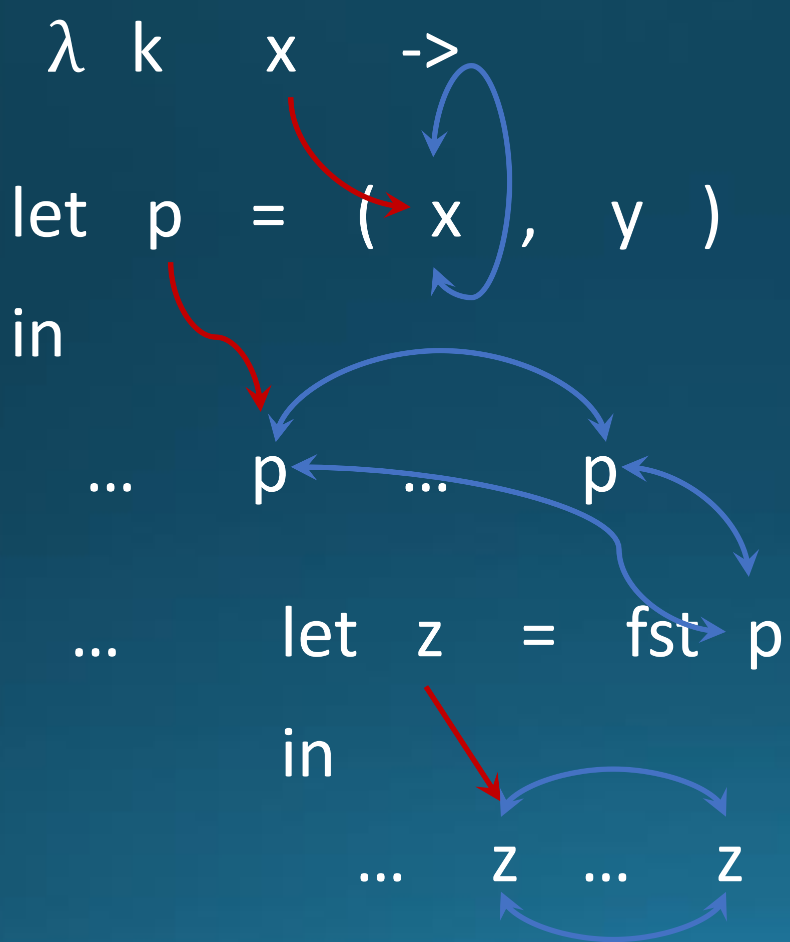
let j z = M in
if c then j a else j b

We already have a join point because every continuation is named

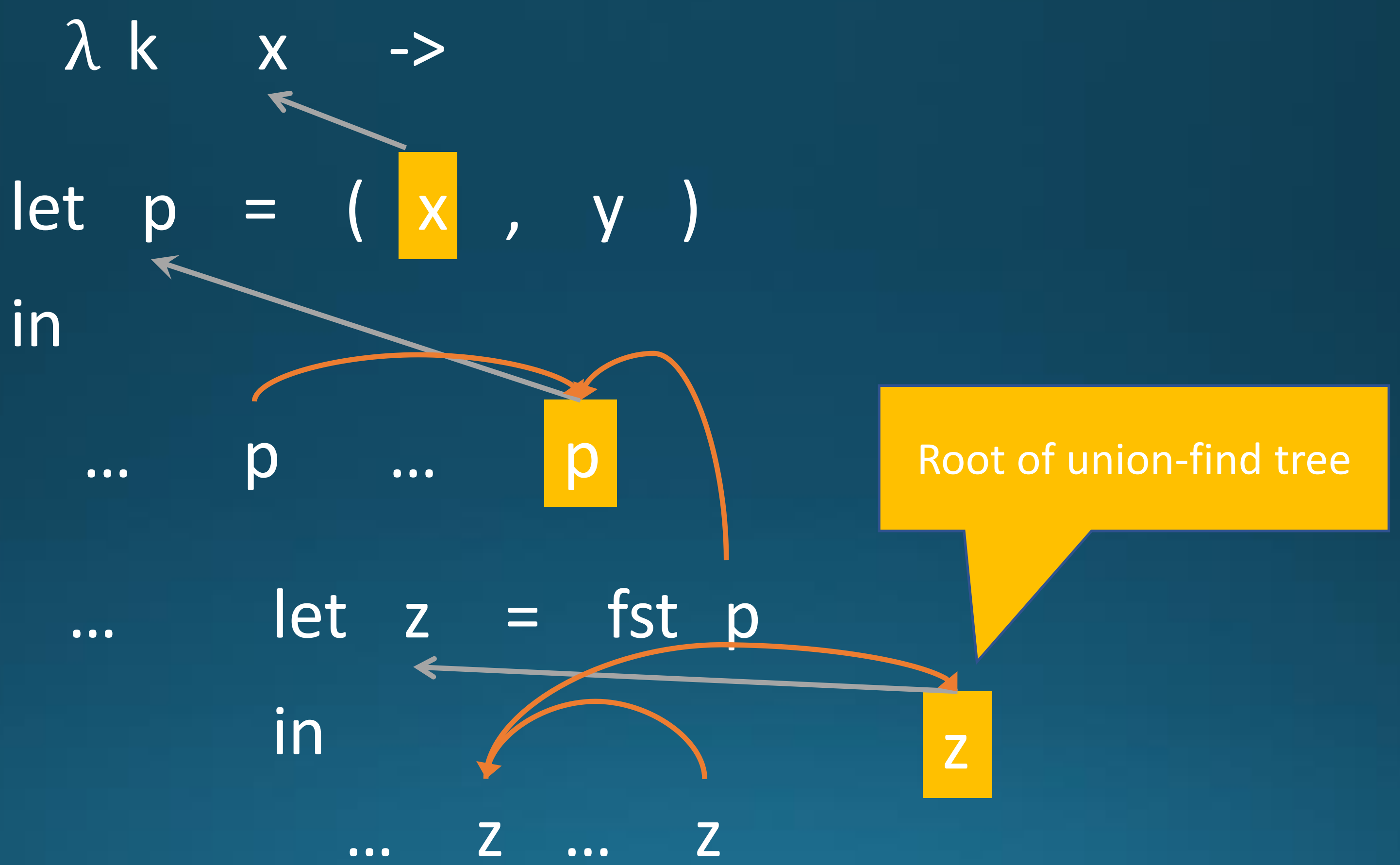
Data structures for CPS IR

- CPS IR can be implemented in functional style: algebraic data type + copy-with-delta transformations
  - For large programs, this is expensive
- Alternative: graph representation + update-in-place
  - Adaptation of idea of Andrew Appel & Trevor Jim
  - Substitution (variable for variable) is constant-time
  - Exhaustive “shrinking” reductions take time linear in size of term
- Three ingredients
  - Doubly-linked tree for basic structure
  - Pointer from bound variable to first free occurrence + doubly-linked circular list between occurrences
  - Union-find data structure to associate occurrences with bound variable

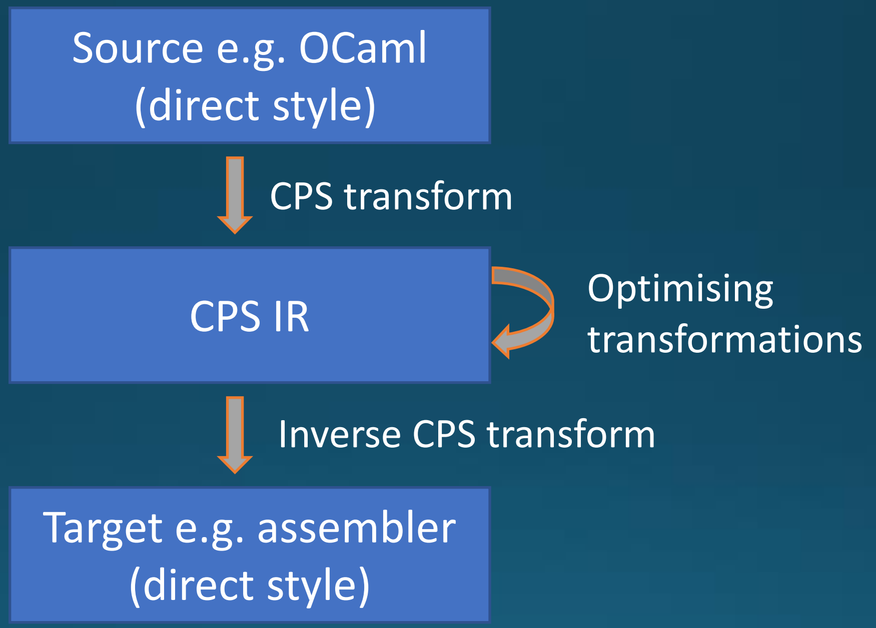
Graph representation
Links from bound to free



Graph representation
Links from free to bound



Compiler pipeline



Contification

```
let f = fun x -> …
in
  g (match z with C c -> f y | D d -> f d)
```

- Function f always returns to the "same place"
  - It can therefore be *contified*: compiled as a code block, with calls compiled as jumps, very efficiently (Fluet & Weeks, 2001)
- Here, it is obvious from the source code
  - For more complex examples, it's not so clear
  - But in CPS IR, it's easy to detect, and to transform general functions into (second-class) continuations

Contification

```
let f = fun x -> ...
in g (match z with C c -> f y | D d -> f d)
```

CPS transform

```
let f = λ k x. ... k ... in
let cont k' w = g r w in
match d with C c -> f k' y | D d -> f k' d
```

Common continuation

contify

```
let cont k' w = g r w in
let cont j x = ... k' ... in
match d with C c -> j y | D d -> j d
```

Hoist k' to bring it into scope

Replace f by continuation j

Substitute actual continuation arg for formal

Contification

- Generalizes to mutually recursive functions
- Really just common-argument elimination
- Iterating this reduction gives *optimal contification* in the sense of Fluet & Weeks (inventors of a dominator-based approach to contification used in the MLton compiler)
- For whole programs, after aggressive optimization (e.g. defunctionalization) a surprising number of functions can be transformed into continuations

Compiler Intermediate Representations (IR)

Functional languages

- Desugared abstract syntax
- Core lambda-calculus-like language, no restrictions
- ANF: Administrative Normal Form (canonical form lambda-calculus)
- Monadic intermediate language
- Continuation-passing style

Close to source

↓

Far from source

Imperative languages

- Desugared abstract syntax
- Flow-graph with local variables
- SSA (Static Single Assignment)
- Continuation-passing style?

Now let's move onto imperative languages

Static Single Assignment form

- Very popular for compiling imperative languages e.g. LLVM
- Every variable is defined before it is used, and assigned exactly once
- At flow-graph "join points", use ϕ pseudo-function to bind variables to values dependent on in-arc to node

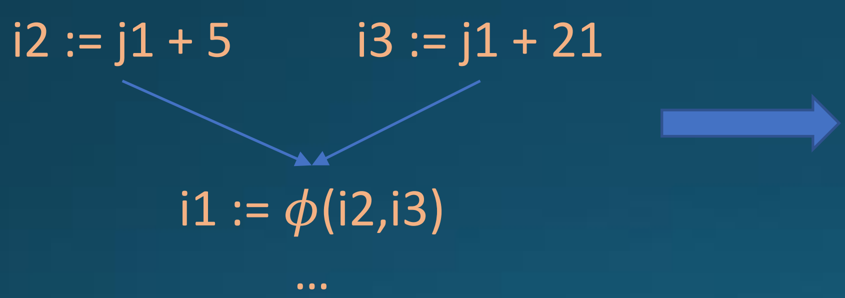
i2 := j1 + 5   i3 := j1 + 21

i1 := ϕ(i2,i3)

…

“SSA is functional programming”

- There’s something odd and upside-down about ϕ nodes.
  - It’s hard to give them clean semantics (though see Damange et al)
  - After function inlining, the “SSA-form” must be recomputed
- Andrew Appel observed “SSA is functional programming”




```
let cont k(i1) = …
in
… let i2 = j1+5 in k(i2)
… let i3 = j1+21 in k(i3)
```

- This fits perfectly in the CPS-based language
  - Function inlining does not destroy well-formedness
  - The “dominance” invariant of SSA is just scoping
  - Loop structure can be expressed using let rec
  - Higher-order code is no problem

Compiling and Typing with Continuations

Andrew Kennedy
Meta London

The Hack programming language

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

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

Static typing of local variables

- No declaration; no declared type; created on first assignment
  - Runtime type typically changes during execution
  - Runtime types can be tested dynamically
- Statically infer types automatically
  - Flow-sensitive
  - At join points, find upper bound of types
  - Type tests *refine* types of locals

Examples of flow sensitivity

```
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**

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

Type error here!

```
function h():void {
  $f = new Foo();
  try {
    bar();
  } catch (Exception $_) {
    $f = new Bar();
  }
  $f->someMethod();
}
```

What type does **$f** have 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

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

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

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

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

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

Typing expressions

- Define a context for locals

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

- For example

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

- Define typing judgment for expressions

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

Typing statements

- Now define a context for continuations,

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

- 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

$$\frac{}{\Gamma; next\!:\Gamma \vdash \{\ \}}$$

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

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

Unreachable: might warn or error

Assignment

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

Conditionals

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

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 typing

- 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 (conditional)

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

Join of environments e.g. union types of locals

Inference (sequencing, assignment)

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

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

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

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

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

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

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)

In practice

- Hack type inference: three techniques
  - Continuations for flow typing
  - Constraint solving for generics and subtyping
  - Some bidirectional type checking for lambdas
- Incremental, parallel, and distributed checking for tens of millions of lines of Hack code, integrated into the IDE

Bibliography

(Introduces ANF)
The Essence of Compiling with Continuations
Flanagan, Sabry, Duba, Felleisen
In *PLDI'93*

Compiling with Continuations, Continued
Andrew Kennedy.
In *ICFP'07*

(For an alternative view!)
Compiling without Continuations
Maurer, Downen, Ariola, Peyton Jones
In *PLDI'17*

Exceptional Syntax
Nick Benton & Andrew Kennedy.
JFP, vol 11 no 4, 2001

SSA is Functional Programming
Andrew Appel. *SIGPLAN Notices*, 33(4), 1998

Contification using Dominators
Matthew Fluet and Stephen Weeks. In *ICFP'01*