Do compilers respect programmers?

William J. Bowman
https://williamjbowman.com
Northeastern University, CCIS
Do compilers respect programmer *invariants*?

William J. Bowman

https://williamjbowman.com
Northeastern University, CCIS
Compilers have one job

Happy source program

Gross machine code
Compilers have one job

Happy source program

abstraction

Gross machine code
Compilers have one job

Abstraction helps programmers design *invariants*: logical assertions that should always hold

Happy source program

abstraction

Gross machine code
Compilers have one job

Happy source program

Abstraction helps programmers design *invariants*: logical assertions that should always hold

- Performance
- Correctness
- Security

Gross machine code
and compilers *don’t do it*

- ✓ Happy source program
- ✗ Gross machine code
Three key problems

Often,

1. Programmers can’t express invariants

2. Compilers don’t respect invariants

3. Linkers don’t enforce invariants
Three key problems

Often,

1. Programmers can’t express invariants

2. Compilers don’t respect invariants

3. Linkers don’t enforce invariants

“invariant”: Belief or intent
Three key problems

Often,

1. Programmers can’t express invariants

2. Compilers don’t respect invariants

3. Linkers don’t enforce invariants
Three key problems

Often,
1. Programmers can’t express invariants
   "invariant": Belief or intent

2. Compilers don’t respect invariants
   In:
   • Optimization
   • Code generation

3. Linkers don’t enforce invariants
   On external code, violating intent
Examples of Problem 1

Often,

1. Programmers can’t express invariants

“invariant”: Belief or intent

2. Compilers don’t respect invariants

   In:
   • Optimization
   • Code generation

3. Linkers don’t enforce invariants

   On external code, violating intent
Programmers can’t express invariants

Example: Mars Climate Orbiter

<table>
<thead>
<tr>
<th>“Invariant”</th>
<th>Ground control calculated <em>Impulse</em> in <em>newton seconds</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Actually</td>
<td><em>Impulse</em> calculated in <em>pound-force seconds</em></td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
</tr>
</tbody>
</table>
Programmers can’t express invariants

Example: Mars Climate Orbiter

<table>
<thead>
<tr>
<th>“Invariant”</th>
<th>Ground control calculated \textit{Impulse} in \textit{newton seconds}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Actually}</td>
<td>\textit{Impulse} calculated in \textit{pound-force seconds}</td>
</tr>
<tr>
<td>\textit{Behavior}</td>
<td>Unscheduled disintegration</td>
</tr>
</tbody>
</table>
Programmers can’t express invariants

Example: Ariana 501

<table>
<thead>
<tr>
<th>“Invariant”</th>
<th>Casts from 64-bit floating point to 16-bit integers do not overflow (and are fast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actually</td>
<td>One did, involving position and acceleration</td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
</tr>
</tbody>
</table>
Programmers can’t express invariants

Example: Ariana 501

<table>
<thead>
<tr>
<th>“Invariant”</th>
<th>Casts from 64-bit floating point to 16-bit integers do not overflow (and are fast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actually</td>
<td>One did, involving position and acceleration</td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
</tr>
</tbody>
</table>
Programmers can’t express invariants

Solutions:
• Types
• Formal methods
• Analyses
Programmers can’t express invariants

Solutions:
- Types
- Formal methods
- Analyses

Possible overflow
Programmers can’t express invariants

Solutions:

- Types
- Formal methods
- Analyses

```haskell
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```
Supposing we *can* express invariants

Definitely, for sure, absolutely holds

- Happy source program

abstraction
Examples of Problem 2

Often,

1. Programmers can’t express invariants

2. Compilers don’t respect invariants

3. Linkers don’t enforce invariants

“invariant”: Belief or intent

In:
- Optimization
- Code generation

On external code, violating intent
Compilers don’t respect invariants

Example: scrubbing secret memory

crypt()
{
    key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}
Compilers don’t respect invariants

Example: scrubbing secret memory

crypt()
{
    key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}

abstraction
Compilers don’t respect invariants

Example: scrubbing secret memory

crypt()
{
    key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory

}"
Compilers don’t respect invariants
Example: scrubbing secret memory

crypt()
{
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0; // scrub memory
}

Compilers don’t respect invariants
Example: scrubbing secret memory

crypt()
{
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0; // scrub memory
}
Compilers don’t respect invariants

Example: scrubbing secret memory

crypt(){
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}

crypt(){
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}
### Compilers don’t respect invariants

**Example: scrubbing secret memory**

```
crypt(){
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}
```

<table>
<thead>
<tr>
<th><strong>Invariant</strong></th>
<th>volatile writes aren’t optimized away</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actually</strong></td>
<td>Depends on the compiler</td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td>Undefined</td>
</tr>
</tbody>
</table>

*“Invariant”*

Compilers don’t respect invariants, with an example:

```
crypt(){
    volatile key = 0xC0DE; // read key
    ... // work with the secure key
    key = 0x0;    // scrub memory
}
```
Compilers don’t respect invariants

Example: floating point precision

\[(10000001.0f \times 10000001.0f) / 10000001.0f = 10000000.0f\]

\[10000001.0f \times (10000001.0f / 10000001.0f) = 10000001.0f\]
Compilers don’t respect invariants

Example: floating point precision

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Compilation doesn’t change floating point precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actually</td>
<td>Depends on the compiler, and the flags</td>
</tr>
<tr>
<td>Behavior</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

\[
(10000001.0f \times 10000001.0f) / 10000001.0f = 10000000.0f
\]

\[
10000001.0f \times (10000001.0f / 10000001.0f) = 10000001.0f
\]
Supposing we *can preserve* invariants

Definitely, for sure, absolutely holds

Definitely, for sure, still holds

abstraction
Examples of Problem 3

Often,

1. Programmers *can’t express invariants*  
   “invariant”: Belief or intent

2. Compilers *don’t respect invariants*  
   In:
   - Optimization
   - Code generation

3. Linkers *don’t enforce invariants*  
   On external code, violating intent
Linkers don’t enforce invariants
Example: Java isn’t memory safe

“Invariant” All Java programs are memory safe

Actually … unless you use the JNI to link with native code

Behavior Undefined
Linkers don’t enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.
Linkers don’t enforce invariants
Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1] 43185 segmentation fault (core dumped)
ocaml verified.ml
```
Linkers don’t enforce invariants
Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)
ocaml verified.ml
```
Linkers don’t enforce invariants
Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)
ocaml verified.ml
```
Linkers don’t enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)
ocaml verified.ml
Linkers don’t enforce invariants

Example: Coq program goes wrong

“Invariant” Coq program verified to be free of out-of-bounds dereferences

Actually … unless you link with anything

Behavior Jumps to arbitrary location in memory

\[ e \rightarrow \text{abstraction} \rightarrow e' \]
What is the behavior when an “invariant” isn’t?
What is the behavior when an “invariant” isn’t?

- Bugs
- Security Attacks
- Undefined Behavior
Goal of my work

- System for expressing invariant
- Invariant-respecting (and exploiting) transformation
- Invariant enforcing linking
My work so far

System for *expressing* invariant

Invariant-*respecting* (and *exploiting*) transformation

Invariant *enforcing* linking

Practical: design & implementation
My work so far

System for *expressing* invariant

Invariant-*respecting* (and *exploiting*) transformation

Foundations: models & techniques

Invariant *enforcing* linking
Building systems to express and exploit invariants:

How do we express and exploit domain-specific performance invariants?

Building systems to express and exploit invariants:

How do we express and exploit domain-specific performance invariants?


How do we more easily express correctness/safety invariants and proofs?

Foundations for preserving and enforcing invariants:
Foundations for preserving and enforcing invariants:

How do we preserve and enforce *language-level invariants (language specs)*?

Foundations for preserving and enforcing invariants:

How do we preserve and enforce *language-level invariants (language specs)*?


How do we preserve and enforce *invariants encoded in dependent types*?

Bowman, Cong, Rioux, Ahmed. *Type-Preserving CPS … is Not Not Possible*. POPL 2018.
Preserving and Enforcing Invariants

System for expressing invariant

Invariant-respecting (and exploiting) transformation

Foundations: models & techniques

Invariant enforcing linking
Preserving Dependent Types

Dependent types

✓ Coq

Invariants and proofs encoded in dependent types
Preserving Dependent Types

- Dependent types
- Type-preserving compiler

From Coq, invariants and proofs encoded in dependent types.

Invariants and proofs preserved.
Preserving Dependent Types

Dependent types

Type-preserving compiler

Safety and correctness invariant enforced at link time.

Coq

 ✓

 Invariants and proofs encoded in dependent types

asm

 ✓

 Invariants and proofs preserved

Type-checking at link time
Why Coq and Dependent Types

- CompCert
- CertiKOS
- Vellvm
- RustBelt
- CertiCrypt

...
Why Coq and Dependent Types

An NSF Expedition
Why Coq and Dependent Types

An NSF Expedition

... which heavily relies on Coq
Why Coq and Dependent Types

DeepSpec is an Expedition in Computing funded by the NSF. … which heavily relies on Coq

> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)
ocaml verified.ml
Encoding invariants in types

Simple types, simple invariants

\[ 5 : \text{Int} \]

\[ / : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ 10 / 5 : \text{Int} \]
More type system features, the more we can encode.

```haskell
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc :: () -> Pound_Force_Seconds

autopilot :: Newton_Seconds -> ()

autopilot (impulse_calc ()) -- Type error!
```
Encoding invariants in types

More type system features, the more we can encode.

```haskell
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```
Encoding invariants in types

More type system features, the more we can encode.

```haskell
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```
Encoding invariants in types

More type system features, the more we can encode.

```haskell
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc :: () -> Pound_Force_Seconds

autopilot :: Newton_Seconds -> ()

autopilot (impulse_calc ()) -- Type error!
```
Encoding invariants in types

Dependent types = Types depend on terms

0 : Int

/ : y:Int -> x:{Int | x != 0} -> Int

10 / [5, 5 != 0] : Int
10 / [0, 0 != 0] -- Type error!
Encoding invariants in types

Dependent types = Types depend on terms

0 : Int

/ : y:Int -> x:{Int | x != 0} -> Int

10 / [5, 5 != 0] : Int
10 / [0, 0 != 0] -- Type error!
Encoding invariants in types

Dependent types = Types *depend on* terms

\[0 : \text{Int}\]

\[/ : y : \text{Int} \rightarrow x : \{\text{Int} \mid x \neq 0\} \rightarrow \text{Int}\]

\[10 / [5, 5 \neq 0] : \text{Int}\]

\[10 / [0, 0 \neq 0] \quad -- \text{Type error!}\]
Encoding invariants in types

Dependent types = Types \textit{depend on} terms

crypt : () \rightarrow () -- \textit{No guarantees}

crypt : () \rightarrow \{\text{key} : \ast \text{Int} \mid \ast \text{key} = 0x0\}
-- \textit{Returns proof invariant was enforced}
Encoding invariants in types

Dependent types = Types depend on terms

\texttt{c \_compiler : C\_AST -> x86\_AST}
Encoding invariants in types

Dependent types = Types depend on terms

\[c\_compiler : \text{C\_AST} \rightarrow \text{x86\_AST}\]
\[c\_interp : \text{C\_AST} \rightarrow \text{Int}\]
\[\text{x86\_interp} : \text{x86\_AST} \rightarrow \text{Int}\]
Encoding invariants in types

Dependent types = Types depend on terms

\[
c_{\text{compiler}} : \text{C\_AST} \rightarrow \text{x86\_AST}
\]

\[
c_{\text{interp}} : \text{C\_AST} \rightarrow \text{Int}
\]

\[
x_{\text{86\_interp}} : \text{x86\_AST} \rightarrow \text{Int}
\]

\[
\text{correct : } x:\text{C\_AST} \rightarrow \\
\qquad c_{\text{interp}}(x) = x_{\text{86\_interp}}(c_{\text{compile}}(x))
\]
Encoding invariants in types

Dependent types = Types depend on terms

c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
         c_interp(x) = x86_interp(c_compile(x))
Encoding invariants in types

Dependent types = Types depend on terms

c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x : C_AST ->
c_interp(x) = x86_interp(c_compile(x))
Encoding invariants in types

Dependent types = Types depend on terms

c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
\[ \text{c_interp}(x) = \text{x86_interp}(\text{c_compile}(x)) \]
Checking dependent types

- Write programs that express proofs

\[
\begin{align*}
c_{\text{compiler}} : & \text{C\_AST} \rightarrow \text{x86\_AST} \\
c_{\text{interp}} : & \text{C\_AST} \rightarrow \text{Int} \\
x_{\text{86\_interp}} : & \text{x86\_AST} \rightarrow \text{Int} \\
\text{correct} : & x : \text{C\_AST} \rightarrow \\
& c_{\text{interp}}(x) = x_{\text{86\_interp}}(c_{\text{compile}}(x))
\end{align*}
\]
Checking dependent types

- Write programs that express proofs
- Refer to code in types

```plaintext
\text{c Compiler:} \quad \text{c Compiler} : \text{C AST} \rightarrow \text{x86 AST}
\text{c Interpreters:} \quad \text{c Interpreter} : \text{C AST} \rightarrow \text{Int}
\text{\text{x86 Interpreter:}} \quad \text{x86 Interpreter} : \text{x86 AST} \rightarrow \text{Int}
\text{Correctness:} \quad \text{correct} : \text{x:C AST} \rightarrow \text{c Interpreter(x) = x86 Interpreter(c Compiler(x))}
```
Checking dependent types

- Write programs that express proofs
- Refer to code in types
- **Decide equality between programs**

\[
\begin{align*}
c_{\text{compiler}} & : \text{C\_AST} \rightarrow \text{x86\_AST} \\
c_{\text{interp}} & : \text{C\_AST} \rightarrow \text{Int} \\
x_{\text{86\_interp}} & : \text{x86\_AST} \rightarrow \text{Int} \\
c_{\text{correct}} & : x : \text{C\_AST} \rightarrow \\
& \quad \text{c\_interp}(x) = \text{x86\_interp}(c_{\text{compile}}(x))
\end{align*}
\]
A type-preserving compiler

- Dependent types
- Type-preserving compiler

Make title case consistent
A type-preserving compiler

CPS: Make control flow explicit

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Morrisett, Walker, Crary, Glew 1998
A type-preserving compiler

Theorem. (Type Preservation)

If \( e : A \)
then translates to
\( e^+ : A^+ \)
A type-preserving compiler

1. Design typed intermediate language
   Prove soundness, decidability, etc
A type-preserving compiler

1. Design typed intermediate language
   Prove soundness, decidability, etc

Hard to design for low-level languages
With *dep. types*, hard to prove sound
A type-preserving compiler

1. Design typed intermediate language
   Prove soundness, decidability, etc

2. Adapt standard translation
   Prove correctness, preservation, etc
A type-preserving compiler

1. Design typed intermediate language
   Prove soundness, decidability, etc

2. Adapt standard translation
   Prove correctness, preservation, etc

Theorem. (Type Preservation)

Contains proofs

\[ \text{if } e : A \text{ then } e^+ : A^+ \]

With dep. types, hard to transform proofs
Brief history of preserving *simple* types

- CPS: Make control flow explicit
- CC: Make data flow explicit
- Alloc: Make allocation explicit
- Code gen: Generate assembly code
Brief history of preserving *simple* types

CPS: Make control flow explicit
1985

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code
Brief history of preserving *simple* types

CPS: Make control flow explicit
1985

CC: Make data flow explicit
1996

Alloc: Make allocation explicit

Code gen: Generate assembly code
Brief history of preserving *simple* types

CPS: Make control flow explicit  
1985

CC: Make data flow explicit  
1996

Alloc: Make allocation explicit  
1998

Code gen: Generate assembly code  
1998
Brief history of preserving *simple* types

**CPS**: Make control flow explicit
- 1985 to 2015

**CC**: Make data flow explicit
- 1996 to 2016

**Alloc**: Make allocation explicit
- 1998 to On-going

**Code gen**: Generate assembly code
- 1998 to On-going

Applications in *secure* compilation
Brief history of preserving dependent types

CPS: Make control flow explicit
1999, 2002

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code
Brief history of preserving dependent types

- CPS: Make control flow explicit
  - 1999, 2002
  - Impossibility result.

- CC: Make data flow explicit

- Alloc: Make allocation explicit

- Code gen: Generate assembly code
Brief history of preserving dependent types

CPS: Make control flow explicit
1999, 2002  POPL 2018  Bowman, Cong, Rioux, Ahmed

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code
CPS for Dependent Types

POPL 2018
Bowman, Cong, Rioux, Ahmed
CPS: Continuation-Passing Style

Makes control flow explicit in syntax.
Equivalent to:
- CFGs
- SSA
CPS: Continuation-Passing Style

```javascript
function f(x) { ... }

f(e)
~>
```
CPS: Continuation-Passing Style

Function f is now a labeled block

```
function f(x) { ... }
f(e)

f : ...+
11: k = l2; e+
12: x = r; goto f
```
CPS: Continuation-Passing Style

\[
\text{function } f(x) \{ \ldots \} \\
f(e) \\
\sim \Rightarrow \\
f : \ldots + \\
l_1: k = l_2; e + \\
l_2: x = r; \text{ goto } f
\]

+ means, recursively translate
CPS: Continuation-Passing Style

```plaintext
function f(x) { ... }

f(e) ~> f

11: k = l2; e+
12: x = r; goto f

Setup calling conventions:
When e+ is finished, store result in r, goto k
```
CPS: Continuation-Passing Style

```plaintext
function f(x) { ... }
f(e)
~>
f : ...+
l1: k = l2; e+
l2: x = r; goto f
```

Setup calling conventions:
When e+ is finished, store result in r, goto k
CPS: Continuation-Passing Style

```plaintext
function f(x) { ... }
f(e)
~> 
f : ...+
l1: k = 12; e+
l2: x = r; goto f
```
CPS: Continuation-Passing Style

A few more technical details

```plaintext
function f(x) { ... } 

f(e) ~> f ...+

main: k = halt; goto l1

l1 : k1 = k; k = l2; e+

l2 : k = k1; x = r; goto f;
```

In `main`, initialize `k` to `special halt`
CPS: Continuation-Passing Style

A few more technical details

```plaintext
function f(x) { ... }
f(e) ~> f :

main: k = halt; goto l1
l1 : k1 = k; k = l2; e+
l2 : k = k1; x = r; goto f;
```

Save/restore “continuation” around translated subexprs
CPS: Continuation-Passing Style

A few more technical details

```plaintext
function f(x) { ... }
f(e)
~>
f : ...
main: k = halt; goto l1
l1 : k1 = k; k = l2; e+
l2 : k = k1; x = r; goto f;
```
CPS: Continuation-Passing Style

5

~>
main: k = halt; goto l1
l1 : r = 5; goto k
CPS: Continuation-Passing Style

5

~>

main: k = halt; goto l1
l1 : r = 5; goto k
CPS: Continuation-Passing Style

5
~>
main: k = halt; goto l1
l1 : r = 5; goto k
CPS: Continuation-Passing Style

...  
x  
~>  
main: k = halt; goto l1  
l1 : k1 = k; k = l2; ...+  
l2 : k = k1; r = x; goto k
CPS: Continuation-Passing Style

...  
x
~>  
main: k = \textbf{halt}; \texttt{goto l1}  
l1 : k1 = k; k = l2; \ldots+  
l2 : k = k1; r = x; \texttt{goto k}
CPS: Continuation-Passing Style

... x

=>

main: k = \textit{halt}; \textit{goto} l1
l1 : k1 = k; k = l2; \textcolor{green}{...+}
l2 : k = k1; r = x; \textit{goto} k
CPS: Continuation-Passing Style

... 

x

~>

main: k = halt; goto l1
l1 : k1 = k; k = l2; ...
+ l2 : k = k1; r = x; goto k
CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>
11: k = 12; e1+
12: x1 = r; k = 13; e2+
13: x2 = r; r = [x1, x2]
```
CPS: Continuation-Passing Style

From now on, ignoring main and save/restore

\[[e_1, e_2]\]

\[\Rightarrow\]

11: \( k = 12; \)  \( e_1^+ \)
12: \( x_1 = r; \)  \( k = 13; \)  \( e_2^+ \)
13: \( x_2 = r; \)  \( r = [x_1, x_2] \)
CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>  
11: k = 12; e1+
12: x1 = r; k = 13; e2+
13: x2 = r; r = [x1, x2]
```
CPS: Continuation-Passing Style

From now on, ignoring **main** and save/restore

\[
[e_1, e_2] \rightarrow
\]

11:  \( k = 12; \ e_1 + \)
12:  \( x_1 = r; \ k = 13; \ e_2 + \)
13:  \( x_2 = r; \ r = [x_1, x_2] \)
Why would CPS be hard?

Invariants about programs ✓
with call/return
Why would CPS be hard?

Invariants about programs ✓
with call/return

Invariants about programs ✓
with \texttt{goto}
Why would CPS be hard?

Invariants about programs with call/return

Invariants about programs with \texttt{goto}

Must design type system to reason about \texttt{goto}
Goal: *Type-preserving* CPS translation

**Theorem. (Type Preservation)**
If $e : A$
then $e^+ : A^+$ translates to
Goal: *Type-preserving* CPS translation

Problem: Σ *types* (strong dependent pairs)

\[
\begin{align*}
p & : \{ x : A \mid B \} \\
\text{snd}(p) & : B[x \mapsto \text{fst}(p)]
\end{align*}
\]
\( p \) is pair of an \( A \) and a \( B \)

\[
p : \{ x : A \mid B \}
\]

\[
snd(p) : B[x \mapsto fst(p)]
\]
\( p \) is pair of an \( A \) and a \( B \)

\[
p : \{x : A \mid B\}
\]

\[
\text{snd}(p) : B[x \mapsto \text{fst}(p)]
\]

where

\( B \) (a type) can refer to

\( x \) (a program var. that stands for \( \text{fst}(p) \) )
\( p \) is a pair of an \( A \) and a \( B \)

\[
\begin{align*}
\ p : & \{ x : A \mid B \} \\
\text{snd}(p) : & B [x \mapsto \text{fst}(p)] \\
\end{align*}
\]

where

\( B \) (a type) can refer to

\( x \) (a program var. that stands for \( \text{fst}(p) \))
Goal: *Type-preserving* CPS translation

\[ p : \{ x : A \mid B \} \]

\[ \text{snd}(p) : B[x \mapsto \text{fst}(p)] \]
Goal: Type-preserving CPS translation

\[
\begin{align*}
 p &: \{x : A \mid B\} \\
\text{snd}(p) &: B[x \mapsto \text{fst}(p)] \\
\end{align*}
\]

11: \( k = l2; \ p^+ \)
12: \( x = r; \ r = \text{snd}(x) \)
Goal: Type-preserving CPS translation

\[ \text{p} : \{x : A \mid B\} \]
\[ \text{snd}(\text{p}) : B[x \mapsto \text{fst}(\text{p})] \]

Need:

11: \( k = l_2; \ p^+ \)
12: \( x = r; \ r = \text{snd}(x) : (B[x \mapsto \text{fst}(p)])^+ \)
Goal: *Type-preserving* CPS translation

\[
p : \{x : A \mid B\} \\
\frac{}{\text{snd}(p) : B[x \mapsto \text{fst}(p)]}
\]

Need:

\[
\begin{align*}
11: & \quad k = l2; \quad p^+ \\
12: & \quad x = r; \quad r = \text{snd}(x) : (B[x \mapsto \text{fst}(p)])^+ 
\end{align*}
\]

Have:

\[
\begin{align*}
11: & \quad k = l2; \quad p^+ \\
12: & \quad x = r; \quad r = \text{snd}(x) : B^+[x \mapsto \text{fst}(x)]
\end{align*}
\]
Goal: *Type-preserving* CPS translation

Suffices:

\[(\text{fst}(p))^+ = \text{fst}(x)\]

\[
p : \{x : A \mid B\}
\]

\[
\text{snd}(p) : B[x \mapsto \text{fst}(p)]
\]

Need:

11: \(k = l_2; p^+\)

12: \(x = r; r = \text{snd}(x) : (B[x \mapsto \text{fst}(p)])^+\)

Have:

11: \(k = l_2; p^+\)

12: \(x = r; r = \text{snd}(x) : B^+[x \mapsto \text{fst}(x)]\)
Need:

11: $k = 12; p$

12: $x = r; r = \text{snd}(x): (B[x \mapsto \text{fst}(p)])^+$

Have:

11: $k = 12; p$

12: $x = r; r = \text{snd}(x): B^+[x \mapsto \text{fst}(x)]$

Suffices:

$(\text{fst}(p))^+ = \text{fst}(x)$
\[ (\text{fst}(p))_+ = \text{fst}(x) \]

11: \( k = l_2; \ p^+ \)
12: \( y = r; \ r = \text{fst}(y) \)
\[(\text{fst}(p))^+ = \text{fst}(x)\]

\begin{align*}
11: \ & k = 12; \ p+ \\
12: \ & y = r; \ r = \text{fst}(y) \\
\end{align*}

ln: ... 
\begin{align*}
& r = v \\
& \text{goto } k
\end{align*}
\[(\text{fst}(p))^+ = \text{fst}(x)\]

11: \(k = 12\); \(p^+\)
12: \(y = r\); \(r = \text{fst}(y)\)

\textbf{Suffices:}

\textbf{Ln:} ... \(r = v\) goto \(k\)

\textbf{Suffices:}

\(x = \text{value-of}(p^+)\)
Suffices:

\[(\text{fst}(p))^+ = \text{fst}(x)\]

\[
\begin{align*}
\text{l1: } & \ k = 12; \ p+ \\
\text{l2: } & \ y = r; \ r = \text{fst}(y)
\end{align*}
\]

But remember, this is \textit{intuitive}; no external code/separate compilation

\text{Suffices:}

\[
x = \text{value-of}(p+)
\]

Barthe and Uustalu 2002
((\text{fst}(p))+) = \text{fst}(x)

11: k = 12; p
12: y = r; r = \text{fst}(y)

But remember, this is \textit{intuitive}; no external code/separate compilation

\textbf{Impossibility result.}

In \textit{standard} CPS, can prove
\[ x \neq \text{value-of}(p+) \]

\textit{Barthe and Uustalu 2002}
\[(\text{fst}(p)) + \]

\[= \text{fst}(x)\]

\[\text{ln: } \ldots \]

\[r = v\]

\[\text{goto } k\]

Impossibility Proof.

If ... can use arbitrary goto, this program can do anything.

If we try to admit this equality, type system must be inconsistent.

Barthe and Uustalu 2002
In implicit invariant to the rescue
Theorem was misunderstood as: In every CPS translation ... Hence this is impossible.

But remember, this is intuitive; no external code/separate compilation

Impossibility result.

In standard CPS, can prove
\[
x \neq \text{value-of}(p+)
\]
$$(\text{fst}(p)) +$$

11: $k = 12$; $p +$
12: $y = r$; $r = \text{fst}(y)$

If ... can use arbitrary goto, this program can do anything.

If we try to admit this equality, type system must be inconsistent.

Impossibility Proof.

Suffices:

$$((\text{fst}(p)) + = \text{fst}(x))$$

ln: ... $r = v$
goto k

Suffices:

$x = \text{value-of}(p +)$

Barthe and Uustalu 2002
Solution

Non-standard, but same control-flow properties

\[ l_1 \colon k = l_2; \quad p^+ \quad \text{Definitely jumps to } l_2 \]

\[ x = \text{value-of}(p^+) \quad \vdash \quad e : B \]

\[ l_1 : k = l_2; \quad p^+ \]
\[ l_2 : x = r; \quad r = e : B \]
Solution

If we can prove this invariant

\[ p^+ : \text{Definitely jumps to } 12 \]
\[ x = \text{value-of}(p^+) \vdash e : B \]

11: \( k = 12; p^+ \)
12: \( x = r; r = e : B \)
If we can prove this invariant

\[ p^+ : \text{Definitely jumps to } 12 \]

\[ x = \text{value-of}(p^+) \]

\[ \vdash e : B \]

Then it's safe to assume this equality

11: \( k = 12; \ p^+ \)

12: \( x = r; \ r = e : B \)
Solution

Develop proof system for this
Develop value interpretation of code with \texttt{goto}
Prove type system consistent

\begin{itemize}
  \item \texttt{p+}: Definitely jumps to \texttt{l2}
  \item \texttt{x = value-of(p+) : B}
  \item \texttt{x = value-of(p+)}
  \item \texttt{l1: k = l2}
  \item \texttt{l2: x = r; r = e : B}
  \item Suffices: \texttt{x = value-of(p+)}
\end{itemize}
Closure Conversion for Dependent Types

PLDI 2018
Bowman, Ahmed
Closure Conversion for Dependent Types

...  
function f(x) {
    return function (y) {
        return x + y;
    }
}

Turn higher-order functions
Closure Conversion for Dependent Types

... function f(x) {
    return function (y) {
        return x + y;
    }
}  

Turn higher-order functions

f(a)(b)
Closure Conversion for Dependent Types

... 
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)

... 
function f(x) {
    return [x, g];
}

function g(x, y) {
    return x + y;
}
c = f(a)
c[2](c[1], b)

Turn higher-order functions into “objects” (closures)
Closure Conversion for Dependent Types

... function f(x) {
    return function (y) {
        return x + y;
    };
}

f(a)(b)

... function f(x) {
    return [x, g];
}

function g(x, y) {
    return x + y;
}

c = f(a)
c[2](c[1], b)

Turn higher-order functions into “objects” (closures)
Closure Conversion for Dependent Types

Turn higher-order functions into “objects” (closures)

1. Decide closure equivalence
2. Transform local proofs into global
X Translation for Dependent Types

1. Decide X equivalence
2. Transform implicit invariants into explicit invariants
Expressing and Exploiting Invariants

System for *expressing* invariant

Invariant- *respecting* (and *exploiting*) transformation

Invariant *enforcing* linking

Practical: design & implementation
Profile-Guided Optimization for DSLs

Domain-specific languages

✓ e
Profile-Guided Optimization for DSLs

- Domain-specific languages
- Optimizing (invariant exploiting) compiler
Suppose you have a pattern when working in some domain

```java
if f(e) {
    ...
} else if g(e) {
    ...
} else if h(e) {
    ...
}
```
Expressing Invariants in DSLs

Maybe you have a lot of such patterns

```javascript
function (k) {
    ... (function (x) {
        ... x k
    })
}
```

```javascript
for (i = 0; i < n; i++){
    f(a[i])
}
```
Expressing Invariants in DSLs

You could create abstractions in a general-purpose lang.

```javascript
function discriminate(e) {
    if f(e) {
        ... } else if g(e) {
            ... } else if h(e) {
                ... }
}
```
Expressing Invariants in DSLs

But maybe in your domain, you know more

```javascript
function discriminate(e) {
    if f(e) {
        e must have certain structure
        ...
    } else if g(e) {
        ...
    } else if h(e) {
        ...
    }
    e must have certain structure
    f(e), g(e), h(e) mutually exclusive
}
```
Expressing Invariants in DSLs

So, you make a new DSL to enforce those invariants

```
route e {
  ...;
  ...;
  ...;
  ...
}
```

checks that `e` is valid at compile-time

knows mutually exclusive
Expressing Invariants in DSLs

Compile to some general-purpose language

```plaintext
code
route e {
  ...;
  ...;
  ...;
}

if f(e) {
  ...
} else if g(e) {
  ...
} else if h(e) {
  ...
}
```

And invariants are lost, can’t be used for optimization
Profile-Guided Optimization

```java
route e {
    b1;
    b2;
    b3;
}
```

1. `route e {
    b1; log("b1");
    b2; log("b2");
    b3; log("b3");
}

2. 

3. execute

4. 

b1 executed 8 times
b2 executed 2 times
b3 executed 102 times
Profile-Guided Optimization

route e {
    b1;
    b2;
    b3;
}

route e {
    b1; log("b1");
    b2; log("b2");
    b3; log("b3");
}

But! PGO can’t use domain invariants (... unless you implement the compiler toolchain yourself)

3. execute

b1 executed 8 times
b2 executed 2 times
b3 executed 102 times
Profile-Guided Optimization

```
route e {
  b1;
  b2;
  b3;
}
```

```
if f(e) {
  b1;
} else if g(e) {
  b2;
} else if h(e) {
  b3;
}
```

1. execute

3. execute

b1 executed 8 times
b2 executed 2 times
b3 executed 102 times
Profile-Guided Meta-Programming

PLDI 2015
Bowman, Miller, St-Amour, Dybvig.
Profile-Guided Meta-Programming

route e {
    b1;
    b2;
    b3;
}

if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}

Meta-programming (macros) for writing DSLs
Profile-Guided Meta-Programming

```
route e {
  b1;
  b2;
  b3;
}
```

```
if f(e) {
  b1;
} else if g(e) {
  b2;
} else if h(e) {
  b3;
}
```

Method for mapping profile back to DSL

- b1 executed 8 times
- b2 executed 2 times
- b3 executed 102 times
Profile-Guided Meta-Programming

route e {
    b1;
    b2;
    b3;
}

if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}

Method for mapping profile back to DSL

Multiple profiles may belong to same DSL expr.
Some profiles may belong to no DSL expr.
Profile-Guided Meta-Programming

route e {
  b1;
  b2;
  b3;
}

if f(e) {
  b1;
} else if g(e) {
  b2;
} else if h(e) {
  b3;
}

Expand, compile and profile

b1 executed 8 times
b2 executed 2 times
b3 executed 102 times
Profile-Guided Meta-Programming

```plaintext
route e {
    b1;
    b2;
    b3;
}

if h(e) {
    b3;
} else if f(e) {
    b1;
} else if g(e) {
    b2;
}
```

Re-expand, using profile to optimize

- **b1** executed 8 times
- **b2** executed 2 times
- **b3** executed 102 times
Profile-Guided Meta-Programming

PLDI 2015
Bowman, Miller, St-Amour, Dybvig.

Implemented for:
• Cisco’s Chez Scheme
• Racket

API, and design sketch for:
• Template Haskell
• MetaOCaml
• Scala
In future

System for *expressing* invariant

Invariant-*respecting* (and *exploiting*) transformation

Invariant *enforcing* linking
In future

- Dependent Types
- Type-preserving compiler
- (Implement) Coq to Dependently Typed Assembly!

Type-checking at link time

don't say "bunch"
Coq to Asm

CPS: Make control flow explicit

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code
Coq to Asm

Foundational challenges:
linear + dependent types
Coq to Asm

Practical challenges
• General purpose
• Low-level optimizations
• Interoperability
Coq to Asm

```plaintext
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)

ocaml verified.ml
```
Coq to Asm

> eeqe verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)

ocaml verified.ml

Type error! Can’t prove \( e' \) safe
OR Types turned into dynamic checks
In future

Dependent Types

Type-preserving compiler

Coq

(Implement) Coq to Dependently Typed Assembly!

Type-checking at link time
In future

- Domain-Specific Dependent Types
- Dependent Types
- Type-preserving compiler

DSLs for dependent types
How do you write sets in your domain?

\[ F = \{ \text{n is an integer} \mid 0 \leq n \leq 19 \} \]

\[ <\text{expr}> ::= <\text{term}>|<\text{expr}><\text{addop}><\text{term}> \]
How do you write sets in Coq

<expr> ::= <term>|<expr><addop><term>

Inductive expr : Set :=
| term_expr : term -> expr.
| add_expr : expr -> term -> expr.
How do you write sets in Cur

\[
<\text{expr}> ::= <\text{term}> | <\text{expr}><\text{addop}><\text{term}>
\]

**Inductive** \( \text{expr} : \text{Set} \) :

\[
| \quad \text{term}_\text{expr} : \text{term} \to \text{expr}. \\
| \quad \text{add}_\text{expr} : \text{expr} \to \text{term} \to \text{expr}. 
\]
How do you write sets in Cur

\[ F = \{ n \text{ is an integer} \mid 0 \leq n \leq 19 \} \]

\[ \text{<expr>} ::= \text{<term>} | \text{<expr>}<\text{addop}><\text{term}> \]

Inductive \text{Set} :=

\[
\begin{align*}
\text{term} & : \text{Set} \\
\text{add_expr} & : \text{expr} \to \text{term} \\
\text{F} & : \{ n \text{ is an integer} \mid 0 \leq n \leq 19 \}
\end{align*}
\]
In future

Domain-Specific Dependent Types

Dependent Types

Type-preserving compiler

Cur

DSLs for dependent types

Coq

asm

e

e'

e+

✓

✓

✓
Long term vision

Expressing and preserving (domain-specific) invariants

Invariant-respecting (and exploiting) compilation

Invariant enforcing linking
Do compilers respect programmers?
Not yet, but here’s a recipe

Expressing and preserving (domain-specific) invariants

Invariant-respecting (and exploiting) compilation

Invariant enforcing linking