Type-Preserving CPS Translation of \( \Sigma \) and \( \Pi \) Types is Not Possible

William J. Bowman, Nick Rioux, Youyou Cong, Amal Ahmed
Type-Preserving CPS Translation of $\Sigma$ and $\Pi$ Types is Not Possible

William J. Bowman, Nick Rioux, Youyou Cong, Amal Ahmed
Dependent types

What are they good for?
Dependent types
What are they good for?

Well.
Dependent types
What are they good for?

Well.
Everything, apparently

Verified in Coq!

- CompCert
- CertiKOS
- Vellvm
- RustBelt
- CertiCrypt
...
Story of a verified program
Story of a verified program

Coq

OCaml

asm

e

e+

...
Story of a verified program

Coq ✓

OCaml ?

Compilation can undo verification

asm ?…?

e

e+

…

e+…+
Compiler correctness!
A correct compilation story

Verify that the program we run is the program we verified
Compiler correctness is not the whole story
Correctness is the “whole program” story
Why type preservation?

Because we do not write whole programs.
Story of a verified component

![Diagram showing verification process from Coq to OCaml with unverified component](image)
Story of a verified *component*

Coq ✓

\[ e : A \]

Compilation can undo verification

OCaml ?

\[ e^+ \]

Linking can undo verification

OCaml ✗

\[ e' \]
Story of a verified component

Coq ✓  OCaml ?  OCaml X  asm ?  asm X  asm X
Story of a verified component

Compilation can undo verification

Linking can undo verification

Linking
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 segmentation fault (core dumped)

ocaml verified.ml
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml

[1] 43185 **segmentation fault** (core dumped)

ocaml verified.ml

Be careful?
Be careful?
No!
Be well-typed!

Verified type-preserving compilers

Coq → ... → Dep. Type ASM

Type checking linkers

e : A

Dep. Type ASM:

e+ : A+
e' : A'
e'' : A''
Great, so let’s preserve dependent types
We motivate the design of a typed assembly language (TAL) and present a type-preservation from System F to TAL. The typed assembly language we present is based on a RISC assembly language, but its static type system provides support for enforcing language abstractions, such as closures, tuples, and user-defined abstract data types. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion, ensures that well-typed programs cannot violate these abstractions. In addition, constructs admit many low-level compiler optimizations.
We motivate the design of a typed assembly language (TAL) and present a type preservation from System F to TAL. The typed assembly language we present is based on a RISC assembly language, but its static type system provides support for enforcing language abstractions, such as closures, tuples, and user-defined abstract data types. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion constructs, admit many low-level compiler optimizations. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion constructs, admit many low-level compiler optimizations.

"No [CPS] translation is possible along the same lines for small Σ-types and sum types with dependent case analysis."
We motivate the design of a typed assembly language (TAL) and present a type-preservation from System F to TAL. The typed assembly language we present is based on a RISC assembly language, but its static type system provides support for enforcing language abstractions, such as closures, tuples, and user-defined abstract data types. The type system ensures that well-typed programs cannot violate these abstractions. In addition, user-defined abstractions admit many low-level compiler optimizations. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion.

“No [CPS] translation is possible along the same lines for small \(\Sigma\)-types and sum types with dependent case analysis.”
not-not is not possible
Why would CPS be hard?

Intuitively, CPS translates every expression of type $A$ into a computation of type $(A \rightarrow \bot) \rightarrow \bot$

i.e. $(\neg \neg A)$
Why would CPS be hard?

Equivalence of expressions

\[ e \equiv e' : A \]

\[ \text{IF} \quad \text{eval}(e) \equiv \text{eval}(e') : A \]
Why would CPS be hard?

Equivalence of expressions

\[ e \equiv e' : A \]
\[ \text{IF} \quad \text{eval}(e) \equiv \text{eval}(e') : A \]

Becomes

Equivalence of computations

\[ c \equiv c' : \neg\neg A \]
\[ \text{IF} \quad c \ ? \equiv c' \ ?' : \bot \]
Why would CPS be hard?

Because deciding equivalence of computations is hard.

Equivalence of expressions

\[ e = e' : A \]
\[ \text{IF } \text{eval}(e) \equiv \text{eval}(e') : A \]

Becomes

Equivalence of computations

\[ c = c' : \neg\neg A \]
\[ \text{IF } c ? \equiv c' ?' : \perp \]
not-not is not possible

now with MATH!
Theorem. (Type Preservation)
If $e : A$
then $e^+ : A^+$ translates to

Goal: *Type-preserving* CPS translation
Goal: *Type-preserving* CPS translation

Problem: For CBN, Σ types (strong dependent pairs)

\[
\frac{\Gamma \vdash e : \Sigma x : A. B}{\Gamma \vdash \text{snd } e : B[fst e/x]} \quad [\text{SND}]
\]
\[ \Gamma \vdash e : \Sigma x : A . B \]

\[ \Gamma \vdash \text{snd } e : B[\text{fst } e / x] \quad \text{[SND]} \]

e is pair of an \textbf{A} and a \textbf{B}
e is pair of an A and a B

\[\Gamma \vdash e : \Sigma x : A. B\]

\[\Gamma \vdash \text{snd } e : B[\text{fst } e/x]\]

where

B (a type) can refer to

x (a term var. standing for (fst e))
$e$ is pair of an $A$ and a $B$

\[
\Gamma \vdash e : \Sigma x : A. B \\
\hline
\Gamma \vdash \text{snd } e : B[\text{fst } e/x]
\]  

where

$B$ (a type) can refer to $x$ (a term var. standing for $(\text{fst } e)$)
Goal: *Type-preserving* CPS translation

\[ \Gamma \vdash e : \Sigma x : A. B \]

\[ \Gamma \vdash \text{snd} \; e : B[\text{fst} \; e / x] \]  

\[ y := e^+; \text{snd} \; y \]

Barthe and Uustalu 2002
Goal: *Type-preserving* CPS translation

\[
\frac{\Gamma \vdash e : \Sigma x : A. B}{\Gamma \vdash \text{snd } e : B[\text{fst } e/x]} \quad \text{[SND]}
\]

Need:

\[
y := e^+; \text{snd } y : B^+[(\text{fst } e)^+/x]
\]
Goal: *Type-preserving* CPS translation

\[
\Gamma \vdash e : \Sigma x : A. B \quad [\text{SND}]
\]

\[
\Gamma \vdash \text{snd } e : B[\text{fst } e/x] 
\]

**Need:**
\[
y := e^+; \text{snd } y : B^+[\text{fst } e/x] 
\]

**Have:**
\[
y := e^+; \text{snd } y : B^+[\text{fst } y/x] 
\]
Goal: *Type-preserving* CPS translation

\[
\Gamma \vdash e : \Sigma x : A. B \quad \text{[SND]}
\]

\[
\Gamma \vdash \text{snd } e : B[\text{fst } e/x]
\]

**Need:**

\[
y := \; e^+; \text{snd } y : B^+[(\text{fst } e)^+/x]
\]

**Have:**

\[
y := \; e^+; \text{snd } y : B^+[\text{fst } y/x]
\]

**Suffices:**

\[
fst y \equiv (\text{fst } e)^+
\]

Barthe and Uustalu 2002
Goal: **Type-preserving CPS** translation

Instead of

\[ y := e^+; \text{snd } y : B^+[(\text{fst } e)^+/x] \]

We have

\[ e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y)) \]

Suffices:

\[ \text{fst } y \equiv (\text{fst } e)^+ \]
Goal: *Type-preserving* CPS translation

Instead of

\[ y := e^+; \text{snd} \; y : B^+[(\text{fst} \; e)^+ / x] \]

We have

\[ e^+ (\lambda \; y : (\sum x : A^+. B^+). (\text{snd} \; y)) \]

To type check a *lambda*, must assume \( y \) is arbitrary

**Suffices:**

\[ \text{fst} \; y \equiv (\text{fst} \; e)^+ \]
Goal: **Type-preserving CPS** translation

Instead of

\[
y := e^+; \text{snd } y : B^+[(\text{fst } e)^+ / x]
\]

We have

\[
e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y))
\]

To type check a lambda, must assume \( y \) is **arbitrary**

and type check the body

Suffices:

\[
\text{fst } y \equiv (\text{fst } e)^+
\]
Need:

\[ y : \sum x : A^+. B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x] \]

\[ e^+ (\lambda y : (\sum x : A^+. B^+). (\text{snd } y)) \]

Suffices:

\[ \text{fst } y \equiv (\text{fst } e)^+ \]
Need:

\[
y : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x]
\]

\[
e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y))
\]

Intuitively,

\[
y \equiv \text{value-of}(e^+)
\]

Suffices:

\[
\text{fst } y \equiv (\text{fst } e)^+
\]
Need:

\[
y : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[ (\text{fst } e)^+/x] \\
e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y))
\]

Suffices:

\[
\text{fst } y \equiv (\text{fst } e)^+
\]

Intuitively,

\[
y \equiv \text{value-of}(e^+)
\]

and

\[
\text{fst } \text{value-of}(e^+) \equiv (\text{fst } e)^+
\]
Need:

\[
y : \sum x : A^+ . B^+ \vdash (\text{snd } y) : B^+[\text{fst } e^+/x]
\]

\[
e^+ (\lambda y : (\sum x : A^+. B^+). (\text{snd } y))
\]

Suffices:

\[
\text{fst } y \equiv (\text{fst } e)^+
\]

Intuitively, \( y \equiv \text{value-of}(e^+) \) and \( \text{fst value-of}(e^+) \equiv (\text{fst } e)^+ \)

Hence

\[
\text{fst } y \equiv (\text{fst } e)^+
\]
Need:

\[ y : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[\text{fst } e^+/x] \]
\[ e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y)) \]

Suffices:

\[ \text{fst } y \equiv (\text{fst } e)^+ \]

Intuitively, and

\[ y \equiv \text{value-of}(e^+) \]
\[ \text{fst } \text{value-of}(e^+) \equiv (\text{fst } e)^+ \]

1. What is the “value-of” a CPS’d computation?
2. How do we remember that value in a continuation?
not not-not is not not not possible
a different CPS
not possible
CPS with answer type polymorphism

Double negation CPS
\[ c : \neg \neg A \]

Polymorphic CPS
\[ c : \forall \alpha. (A \rightarrow \alpha) \rightarrow \alpha \]
CPS with answer type polymorphism

Double negation CPS
\[ c : \neg \neg A \]
\[ c \left( \lambda \left( v \right) \ldots \right) : \bot \]

Polymorphic CPS
\[ c : \forall \alpha. \ (A \rightarrow \alpha) \rightarrow \alpha \]
\[ c \ B \left( \lambda \left( v \right) \ldots \right) : B \]
CPS with answer type polymorphism

Double negation CPS
\[ c : \neg \neg A \]
\[ c (\lambda (v) \ldots) : \bot \]

Polymorphic CPS
\[ c : \forall \alpha. (A \rightarrow \alpha) \rightarrow \alpha \]
\[ c \ B (\lambda (v) \ldots) : B \]
1. The value of a CPS’d computation
The key is equivalence

Before

\[ c \equiv c' : \neg\neg A \]

IF \[ c \equiv c' ?' : \bot \]
The key is equivalence

Before

\[ c \equiv c' : \neg\neg A \]

IF \[ c \not\equiv c' ? ?' : \bot \]

After

\[ c \equiv c' : \forall \alpha. (A \rightarrow \alpha) \rightarrow \alpha \]

IF \[ c A \ id \equiv c' A \ id : A \]
The key is equivalence

Before

\[ c \equiv c' : \neg\neg A \]

\[ \text{IF} \quad c ? \equiv c' ?' : \bot \]

Meaningless garbage

After

\[ c \equiv c' : \forall \alpha. (A \to \alpha) \to \alpha \]

\[ \text{IF} \quad c \ A \ id \equiv c' \ A \ id : A \]

Meaningful underlying *value* of type \( A \)
Add to our typed CPS target language

\[ \Gamma \vdash (e_1 \ B \ (\lambda x : A. \ e_2)) \equiv (\lambda x : A. \ e_2) \ (e_1 \ A \ \text{id}) \]  

[\equiv\text{-Cont}]

Based on
“continuation shuffling”
aka
“parametricity condition”
aka
“naturality”
When computation $e_1$
When computation $e_1$ is applied to answer type $B$ and continuation

\[ \Gamma \vdash (e_1 \ B (\lambda x : A. e_2)) \equiv (\lambda x : A. e_2) (e_1 \ A \ id) \]
When computation $e_1$ is applied to answer type $B$ and continuation

\[ \Gamma \vdash (e_1 \ B \ (\lambda \ x : A. \ e_2)) \equiv (\lambda \ x : A. \ e_2) \ (e_1 \ A \ \text{id}) \] [\equiv\text{-Cont}]

“Shuffle” this into continuation, as a function
When computation $e_1$ is applied to answer type $B$ and continuation

\[
\Gamma \vdash (e_1 \ B \ (\lambda x : A. \ e_2)) \equiv (\lambda x : A. \ e_2) \ (e_1 \ A \ id)
\]

“Shuffle” this into continuation, as a function, applied to value of $e_1$
\[
\Gamma \vdash (e_1 B (\lambda x : A. e_2)) \equiv (\lambda x : A. e_2)(e_1 A \text{id})
\]

value-of(e_1) \overset{\text{def}}{=} e_1 A \text{id}
Modeling $\equiv$-Cont

- Parametricity Translation

$\text{CC w/ } \equiv$-Cont $\rightarrow$ Extensional CC

Prove extensional $\equiv$-Cont as a free theorem.
Towards Type Preservation

**Need:**

\[
y : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x]
\]

\[
e^+(\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y))
\]

Intuitively,

\[
y \equiv e^+ \text{id}
\]

**Suffices:**

\[
\text{fst } y \equiv (\text{fst } e)^+
\]

\[
\text{and}
\]

\[
\text{fst } (e^+ \text{id}) \equiv (\text{fst } e)^+
\]
Towards Type Preservation

\[
(fst\ e)^+ = e^+ (\lambda y.\ fst\ y)
\]

By ≡-Cont

\[
≡ (\lambda y.\ fst\ y) (e^+ \ id)
\]

≡ \(fst (e^+ \ id)\)

Intuitively,

\[
y ≡ e^+ \ id
\]

\[
fst (e^+ \ id) ≡ (fst\ e)^+
\]
1. The value of a CPS’d computation

\[ \Gamma \vdash (e_1 \ B \ (\lambda \ x : A. \ e_2)) \equiv (\lambda \ x : A. \ e_2) \ (e_1 \ A \ id) \]

\[ \text{value-of}(e_1) \overset{\text{def}}{=} e_1 \ A \ id \]
2. Remember value when jumping to a continuation
Need:

\[ y : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+ / x] \]

\[ e^+ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y)) \]

Suffices:

\[ \text{fst } y \equiv (\text{fst } e)^+ \]
Intuitively,

\[ y \equiv \text{value-of}(e^+) \]
Need:
\[ y : \text{Σ } x : A^+ . B^+ \vdash (\text{snd } y) : B^+[\left(\text{fst } e\right)^+/x] \]
\[ e^+ \left( \lambda y \cdot (\text{Σ } x : A^+ . B^+). (\text{snd } y) \right) \]

Intuitively,
\[ y \equiv \text{value-of}(e^+) \]

Suffices:
\[ \text{fst } y \equiv \left(\text{fst } e\right)^+ \]

But formally, \( y \) is arbitrary.
What if we make this equivalence

\[ y \equiv \text{value-of}(e^+) \]

<table>
<thead>
<tr>
<th>y : ( \Sigma x : A^+. B^+ )</th>
<th>( \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e^+ ) ( \lambda y : (\Sigma x : A^+. B^+). (\text{snd } y) )</td>
<td></td>
</tr>
</tbody>
</table>

part of this typing rule
$y = e^+ \text{id} : \Sigma x : A^+.B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x]$

$e^+ @ (\lambda y : (\Sigma x : A^+.B^+). (\text{snd } y))$

“Jump-to” form
Jumping to a continuation is not just application.
Jumping to a continuation is not just application.

\[ \Gamma \vdash e : \Pi \alpha : \ast. (A \to \alpha) \to \alpha \quad \Gamma \vdash B : \ast \quad \Gamma, x = e \ A \ \text{id} \vdash e' : B \]

\[ \Gamma \vdash e \ @ \ B (\lambda x : A. \ e') : B \]
Jumping to a continuation is not just application.

\[\Gamma \vdash e : \Pi \alpha : \ast. (A \to \alpha) \to \alpha \quad \Gamma \vdash B : \ast \quad \Gamma, x = e A \text{ id} \vdash e' : B\]

\[\Gamma \vdash e @ B (\lambda x : A. e') : B\]
Modeling T-Cont

CC^K

CC w/ \equiv\text{-Cont} w/ T-Cont

ANF translation

Parametricity Translation

CC w/ \equiv\text{-Cont}

Extensional CC

Prove extensional \equiv\text{-Cont} as a free theorem.
2. Remember value when jumping to a continuation

\[
\Gamma \vdash e : \Pi \alpha : \ast. (A \to \alpha) \to \alpha \quad \Gamma \vdash B : \ast \quad \Gamma, x = e A \text{id} \vdash e' : B
\]

\[
\Gamma \vdash e \mathrel{@} B (\lambda x : A. e') : B
\] [T-Cont]
Need:
\[ y = e^+ \text{id} : \Sigma x : A^+. B^+ \vdash (\text{snd} \ y) : B^+[(\text{fst} \ e)^+/x] \]
\[ e^+ \ @ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd} \ y)) \]

Suffices:
\[ \text{fst} \ y \equiv (\text{fst} \ e)^+ \]
Type Preservation

Need:

\[ y = e^+ \text{id} : \Sigma x : A^+. B^+ \vdash (\text{snd} y) : B^+[(\text{fst} e)^+/x] \]

\[ e^+ \ @ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd} y)) \]

By \( \text{T-Cont} \),

\[ y \equiv e^+ \ \text{id} \]

Suffices:

\[ \text{fst} y \equiv (\text{fst} e)^+ \]
Type Preservation

Need:

\[
y = e^+ id : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[(\text{fst } e)^+/x]
\]

\[
e^+ @ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y))
\]

By T-Cont,

\[
y \equiv e^+ id
\]

Suffices:

\[
fst y \equiv (\text{fst } e)^+
\]

By earlier (uses ⇑-Cont)

\[
fst (e^+ id) \equiv (\text{fst } e)^+
\]
Type Preservation

Need:

\[ y = e^+ \ id : \Sigma x : A^+. B^+ \vdash (\text{snd } y) : B^+[ (\text{fst } e)^+/x] \]

\[ e^+ @ (\lambda y : (\Sigma x : A^+. B^+). (\text{snd } y)) \]

By T-Cont,

\[ y \equiv e^+ \ id \]

Hence

\[ \text{fst } y \equiv (\text{fst } e)^+ \]

Suffices:

\[ \text{fst } y \equiv (\text{fst } e)^+ \]

By earlier (uses \equiv-Cont)

\[ \text{fst } (e^+ \ id) \equiv (\text{fst } e)^+ \]
Call-by-value
CBV CPS fails, even for $\Pi$

\[
\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 \ e_2 : B[e_2/x] \\
\end{array}
\]

[App]
CBV CPS fails, even for $\Pi$

\[ \Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A \]

\[ \frac{}{\Gamma \vdash e_1 e_2 : B[e_2/x]} \text{ [App]} \]

**Need:**

\[ y_1 := e_1^+; y_2 := e_2^+; y_1 y_2 : B^+[e_2^+/x] \]

**Have:**

\[ y_1 := e_1^+; y_2 := e_2^+; y_1 y_2 : B^+[y_2/x] \]
But $\Pi$ is fine is CBN

\[
\begin{align*}
\Gamma \vdash e_1 : \Pi x : A. B & \quad \Gamma \vdash e_2 : A \\
\hline
\Gamma \vdash e_1 \ e_2 : B[e_2/x] & \quad \text{[App]} \\
\end{align*}
\]

**CBV:**
\[
\begin{align*}
y_1 & := e_1^+; \\
y_2 & := e_2^+; \\
y_1 \ y_2 & : B^+[e_2^+/x]
\end{align*}
\]

**CBN:**
\[
\begin{align*}
y_1 & := e_1^+; \\
y_1 \ e_2^+ & : B^+[e_2^+/x]
\end{align*}
\]

But $\Pi$ is fine is CBN
Compiler Correctness, Too
Theorem. (Correctness of Separate Compilation)

If $e \gamma$ then $v'$ $\equiv v^+$
Lemma. (Equivalence Preservation)

If \( e \equiv e' \)

then \( e^+ \equiv e'^+ \)

Lemma. (Preservation of Reduction Sequences)

If \( e \rightarrow e' \)

then \( e^+ \rightarrow e'' \equiv e'^+ \)
Scaling to Coq
Our compiler so far

CC (w/ \( \Sigma \))

CPS

CC^K
Future Work

CC (w/ Σ) → CC^K

CPS

Coq

Coq^K

CoqASM
Future Work

CC (w/ $\Sigma$) \[\xrightarrow{\text{CPS}}\] CC$^K$ \[\xrightarrow{\text{Lot o’ work}}\] Coq$^K$ 

CoqASM
In 2002, Barthe and Uustalu also prove:

- *No type-preserving CPS translation can exist* with dependent case analysis on sum types.
Dependent Case

In 2002, Barthe and Uustalu also prove:

• *No type-preserving CPS translation* can exist with dependent case analysis on sum types.

* that admits call/cc in certain contexts.
Dependent Case

In 2002, Barthe and Uustalu also prove:

- No type-preserving CPS translation* can exist with dependent case analysis on sum types.

Answer type polymorphism disallows call/cc, in any compiled code.

* that admits call/cc in certain contexts.
In 2002, Barthe and Uustalu also prove:

- *No type-preserving CPS translation* can exist with dependent case analysis on sum types.

Answer type polymorphism disallows call/cc, in *any* compiled code.

We sketch extending our translation to dependent case

* that admits call/cc in certain contexts.
Type-Preserving CPS Translation of $\Sigma$ and $\Pi$ Types is Not Possible

williamjbowman.com/#cps-sigma

1. The “value-of” a CPS’d computation and
2. “remember” that value when jumping to a continuation.