

Compiling with Dependent Types

William J. Bowman



- Hello everyone.
- I want to tell you something about compiling with dependent types.
-

Dependent Types

What are they good for?

- Let me start with dependent types.
- We've all probably heard of them; 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
- ...

- just to name a few...
- verified compilers, operating systems, languages semantics, and crypto stuff, etc.
- Turns out, useful for high-assurance software, to simultaneously develop, specify, prove
- and machine-checked your work

Story of a verified program

Coq



So that's great, we have lots of verified code. Verified... in Coq.

But that's not the end of the story.

Story of a verified program



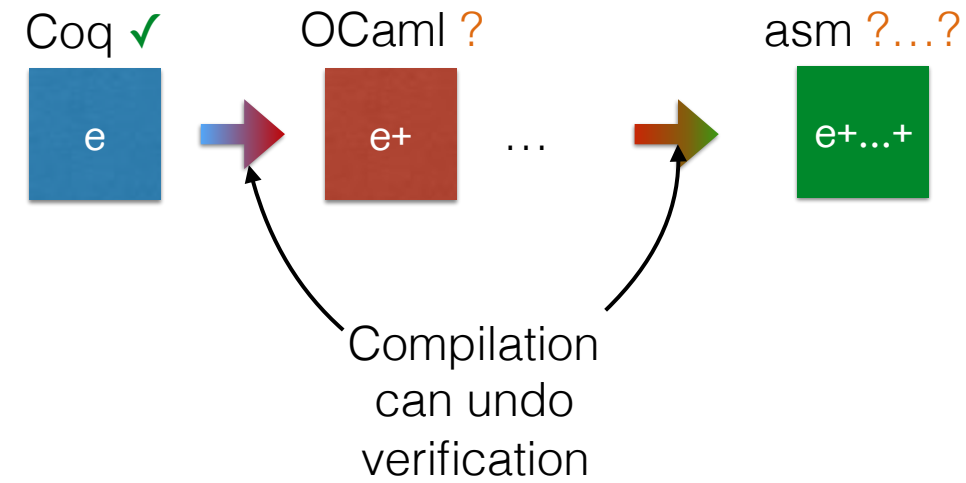
The life of a verified program is just beginning.

Because hopefully, after writing some verified code, you'd like to run it.

After verifying in Coq, we now extract (compile) to OCaml, perhaps.

Then further to assembly.

Story of a verified program



And each step might introduce ... bugs.

Each step of compilation is an opportunity to undo all our verification efforts.



Compiler correctness!

A correct compilation story



Verify that the program we run is the program we verified

Ah yes, we should prove each stage of compilation correct.

Then we know that the program that ends up running is verified to be the program we verified

That's a good story.

Compiler correctness
is not the whole story

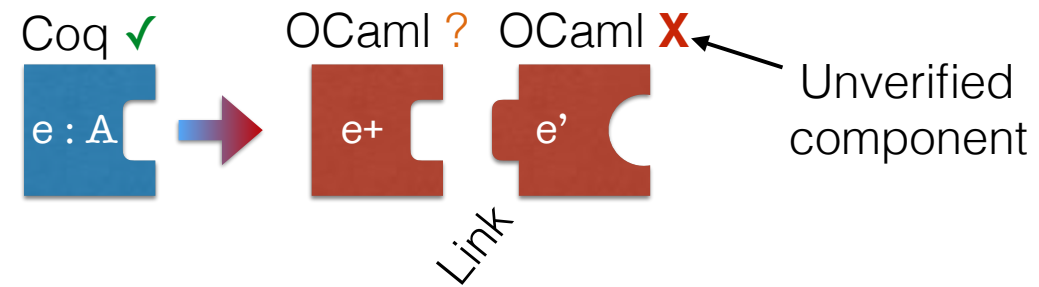
Correctness is the
“whole program” story

So we need compiler correctness,
But we need something else...

We do not write whole programs.

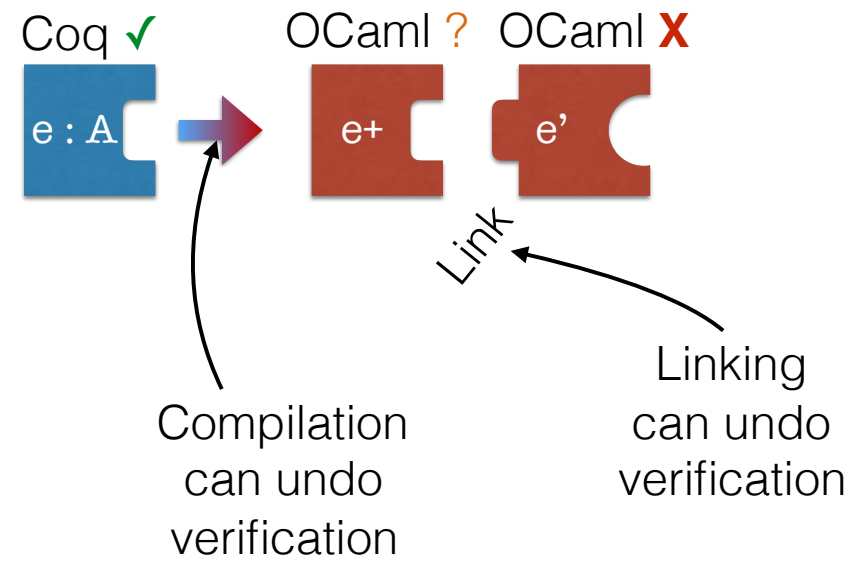
because even a verified compiler can end up producing incorrect code, if we compile and then link with ANYTHING
And no one writes whole programs.

Story of a verified *component*

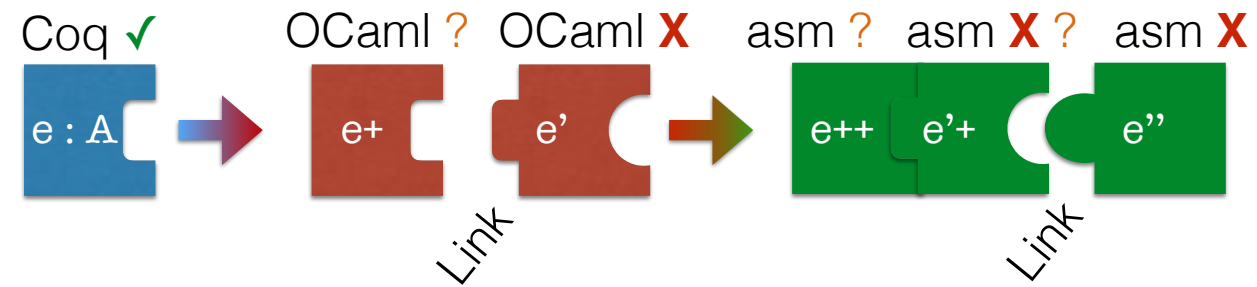


- Most programs are not whole programs.
- Program will be compiled, then **linked** w/ external components

Story of a verified *component*

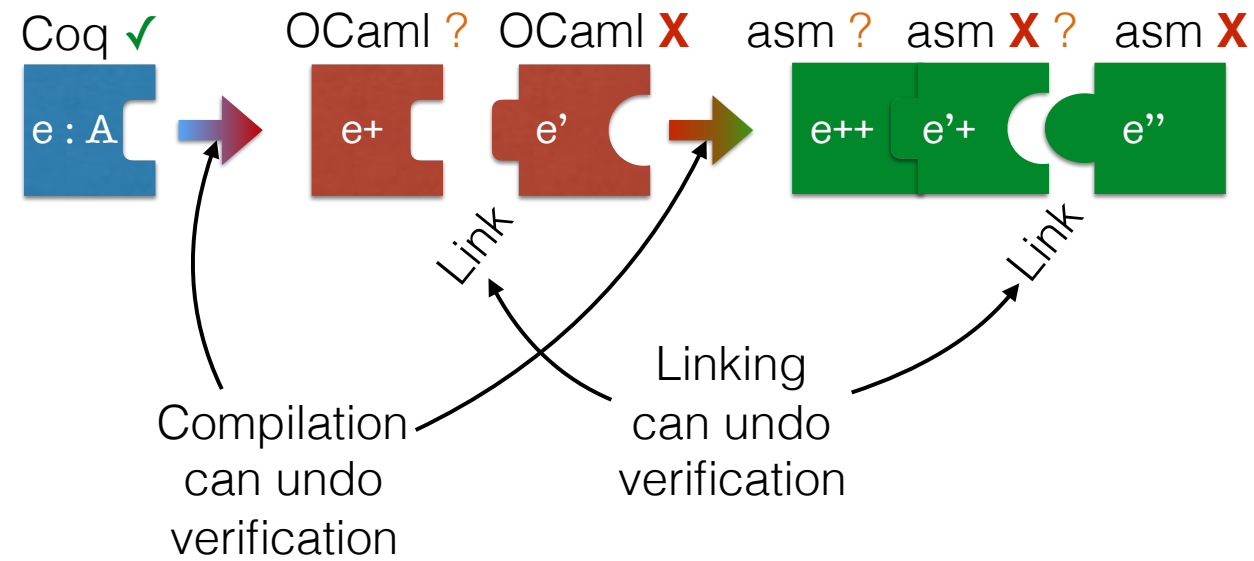


Story of a verified *component*



- (e.g. w/ OCaml, runtime, OS, low-level C libraries)

Story of a verified *component*



- (e.g. w/ OCaml, runtime, OS, low-level C libraries)

A real verified Coq program



```
> coqc verified.v
```

```
> link verified.ml unverified.ml
```

```
> ocaml verified.ml
```

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

```
ocaml verified.ml
```

- This is no hypothetical

linked with **1 line** of unverified code

```
> coqc verified.v
```

```
> link verified.ml unverified.ml
```

```
> ocaml verified.ml
```

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

```
ocaml verified.ml
```

- I can link with 1 line of unverified code, which could pass for honest well-typed OCaml code.

```
> coqc verified.v  
  
> link verified.ml unverified.ml  
  
> ocaml verified.ml  
[1] 43185 segmentation fault (core dumped)  
ocaml verified.ml
```

Jumps to arbitrary location in memory



- And get this verified program to jump to an arbitrary location in memory.
- ... and do you know what the state-of-the-art is?

Be careful?

```
> coqc verified.v
```

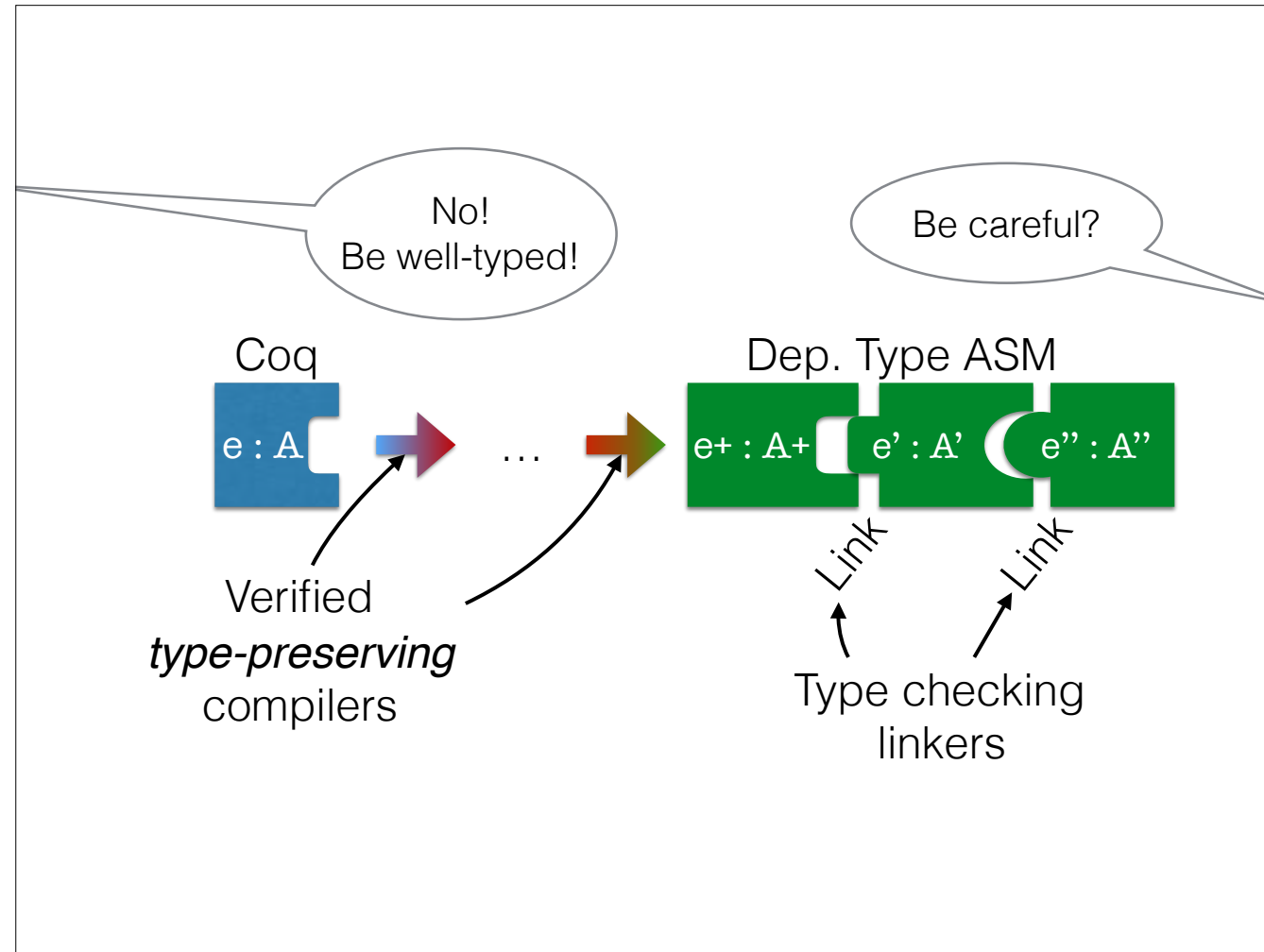
```
> link verified.ml unverified.ml
```

```
> ocaml verified.ml
```

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

```
ocaml verified.ml
```

- The state of the art in avoiding this... is telling the programmer to be careful when linking.
- All verification effort, all our careful static checking... WASTED
- Just to tell ourselves we're smart enough to keep all our invariants in our head while linking



- No i say!
- I'm not smart enough to be careful, and frankly I don't trust most programmers much either.
- To get guarantees at link time, we can preserve the types, and use them when linking.
- The grand vision, is to compile to a dependently typed assembly.
- Once more use types to enforce our invariants.

Great, so let's
preserve dependent types

Great so everyone is on board; let's preserve ourselves some dependent types.

Type Preservation 101

So. How do we do that?

Well thankfully, there's a recipe.

Type Preservation 101

From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER

Cornell University

KARL CRARY

Carnegie Mellon University

and

NEAL GLEW

Cornell University

We motivate the design of a *typed assembly language* (TAL) and present a type-preserving translation 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, the constructs admit many low-level compiler optimizations. Our translation to TAL is as a sequence of type-preserving transformations, including CPS and closure conversion.

And the recipe is in this paper, system F to typed assembly language.

And ever since, everyone working on type preservation follows this recipe.

Type Preservation 101

From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER

Cornell University

KARL CRARY

Carnegie Mellon University

and

NEAL GLEW

Cornell University

“a sequence of type-preserving transformations,
including CPS and closure conversion...”

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, the constructs admit many low-level compiler optimizations. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion.

And the recipe begins with two key compiler passes: CPS and closure conversion.

Type Preservation 101

From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER

Cornell University

KARL CRARY

Carnegie Mellon University

and

NEAL GLEW

Cornell University

High-level reasonable code into
low-level machine-friendly code

“a sequence of type-preserving transformations,
including CPS and closure conversion...”

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, the constructs admit many low-level compiler optimizations. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion.

These two passes are really responsible for the taking high-level function features.

And encoding them in a low-level machine-friendly encoding.

The rest of the compiler is really all about uninteresting nonsense, like the particulars of machine words, or the number of registers, or making things go fast.

So let's just.. apply the recipe to a dependently typed language?

Type Preservation 101

From System F to Typed Assembly Language

GRE CPS Translating Inductive and Coinductive Types

Corr

KAF

Carn

and

NEA

Cornell University

Gilles Barthe

[Extended Abstract]

Tarmo Uustalu*

“**No** [CPS] translation *is possible* along the same lines for small Σ -types and sum types with dependent case analysis.”

ing constructs admit many low-level compiler optimizations. Our translation to TA as a sequence of type-preserving transformations, including CPS and closure conve

That’s a good idea. In fact, it’s such a good idea, someone already tried.

And they found... actually.... it’s impossible.

At least once we add actual features found in actual dependently typed languages.

My thesis

Type-preserving compilation of dependently typed languages is a viable technique for statically eliminating classes of errors introduced during compilation and linking.

Well, actually,

“~~No~~[CPS] translation *is possible* along the same lines for small Σ -types and sum types with dependent case analysis.”

which brings me to my thesis.

- But we already did the proposal part.

My thesis

Type-preserving compilation of dependently typed languages is a viable technique for statically eliminating classes of errors introduced during compilation and linking.

Subset of Coq:
Calculus of Constructions (CC), with Σ -types,
dependent case analysis, higher universes.

Now, I've actually shown it.

And the key definition here is languages, and viable.

I want this to scale to languages that actually get used in practice, not small core calculi.

So I want this for a subset of Coq that includes all the core features of dependent types; and I'll discuss these a bit shortly.

Roadmap

Proving that thesis involves a lot of formal work.

Roadmap

In this talk: 1 central lesson

In this talk, I'm going to present a one central lesson.

Roadmap

In this talk: 1 central (formal) lesson

And it's a very formal lesson.

Roadmap

In this talk: 1 central (formal) lesson

How do we model machine computations?

That less is: how do we model computation

Roadmap

In this talk: 1 central (formal) lesson

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

That less is: how do we model computation

Roadmap

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

This is very formal work, and I need to show you formalism to communicate this.

I'm not just trying to develop a compiler, nor just convincing you the compiler is correct.

I need to design a compiler that can convince a type checker that it is correct.

A lot of machine-reasoning; means a lot of formal.

Roadmap

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

1. Dependent types
2. Sequencing Computations
3. Relocating Computations
4. Lessons

Roadmap

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

1. Dependent types
 2. Sequencing Computations
 3. Relocating Computations
 4. Lessons
- 
- Dependency*—key to source reasoning

Roadmap

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

1. Dependent types
2. Sequencing Computations
3. Relocating Computations
4. Lessons

Dependency—key to
source reasoning

Study CPS &
dependency

Second, I will walk through a study how we model sequences of computations.

Typically, this is done via CPS, so we'll student how CPS interact with dependency.

But the lesson I hope to show you is that underlying CPS is the need to model dependent sequences of computations

Roadmap

How do we model machine computations?

- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

1. Dependent types
2. Sequencing Computations
3. Relocating Computations
4. Lessons

Dependency—key to
source reasoning

Study CPS &
dependency

Study closure
conversion

Third, I'll look at how we model computations that can be moved out of their current scope.

This is typically presented as closure conversion, so we'll study how it interact with dependency.

But the lesson is that that past focus on closure conversion distracts us from focusing on modeling the machine idea: a computation that a compiler can move into a different scope.

Roadmap

How do we model machine computations?

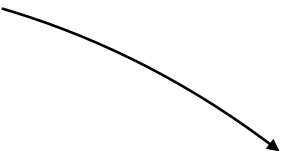
- preserving *all* source reasoning
- in a decidable, *machine-verifiable* way

1. Dependent types
2. Sequencing Computations
3. Relocating Computations
4. Lessons

Dependency—key to
source reasoning

Study CPS &
dependency

Study closure
conversion



Summary
Proof recipe

Core Elements of Dependent Types

(6 min)

so if we're going to be compiling dependent types, we need to understand them.

To design new dependent types systems for compiler intermediate language, we need to understand them formally.

First, Simple Types

syntax

<i>Terms</i>	e	$::=$	
<i>Types</i>	A, B	$::=$	

As review, we all know about typed languages. Here's a typed language, formally.

We have some terms, written with meta-variable e

And some types, written with A and B

Simple Types

syntax

Terms e ::= x |

Types A, B ::=

Terms include things like variables.

Simple Types

syntax

Terms e ::= x | $true$ | $false$ |

Types A, B ::= $bool$ |

And the boolean values true and false.

Types include things like, bool, the type of booleans.

Simple Types

syntax

Terms e ::= x | true | false | $\lambda x. e$ |

Types A, B ::= bool | $A \rightarrow B$ | $\dots$

We also have functions, $\lambda x. e$

And the function type, $A \rightarrow B$

Simple Types

syntax

Terms e ::= $x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

Types A, B ::= $\text{bool} \mid A \rightarrow B \mid \dots$

We can apply function, e_1 applied to e_2 .

Simple Types

typing rules

Terms $e ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

Types $A, B ::= \text{bool} \mid A \rightarrow B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B}$$

Then we have typing rules.

Terms, like functions, have types, like $A \rightarrow B$.

Simple Types

typing rules

Terms $e ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

Types $A, B ::= \text{bool} \mid A \rightarrow B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B}$$

With a function, we type check the body e under the assumption that x has type A .

Simple Types

typing rules

Terms $e ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

Types $A, B ::= \text{bool} \mid A \rightarrow B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B}$$

$$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B}$$

When we apply a function, e_1 to e_2 , we check that e_1 is a function and e_2 is well-typed.

Simple Types

typing rules

Terms $e ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

Types $A, B ::= \text{bool} \mid A \rightarrow B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B}$$

$$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B}$$

And we check that the argument's type is equal to the expected argument type.

Here I've been extra pedantic and written that the types A and A' must be equivalent, rather than just using the same variable name.

With just simple types, this equivalence is trivial; just syntactic equivalence.

Now, Dependent Types

syntax

Terms $e ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \mid \dots$

$$\text{Types } A, B ::= \text{bool} \mid A \rightarrow B \mid \dots$$

$$\begin{array}{lcl} \textit{Expressions} & e, A, B & ::= \quad x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2 \\ & & \mid \text{bool} \mid A \rightarrow B \mid \dots \end{array}$$

So now dependent types, by contrast.

First, the syntax changes. Terms and types appear in the same syntax. You can compute with types, or refer to terms in types.

Now, Dependent Types

syntax

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid A \rightarrow B \mid \dots$



Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

Ooohh, Greek!

But this isn't enough. To really use dependent types, we need the types themselves to change. Types should be able to name and refer to terms. For example, the function type changes. Now instead of a boring $A \rightarrow B$, it's a greek letter, so you know it's powerful. And the result type, B , can refer to the argument x by name, to do things like express a pre or post condition about that argument x .

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$

Typing rules get weird. So function is about what you would expect.

A function has this pi type when the body e is well-typed under the assumption that x has type A .

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$

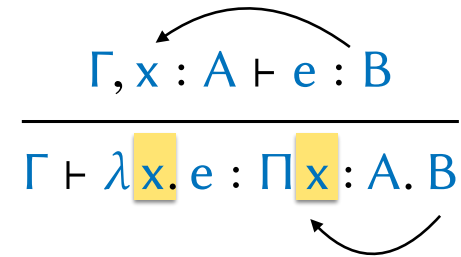
Except now the name x is bound in both the function and in its type.

This means we can express pre and post conditions on functions.

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$


$\text{div} : \Pi x : \text{Int}. \Pi y : \text{Int}. \Pi p : y \neq 0. \text{Int}$

Like maybe we want division to be a function that takes two integers, x and y.

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$

$\text{div} : \Pi x : \text{Int}. \Pi y : \text{Int}. \Pi p : y \neq 0. \text{Int}$

And a proof that we're not going to try to divide by zero.

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Weird things happen especially when we *use* a dependent type, like in application.

As usual, we check that e_1 is a function and e_2 is an argument, and the type of e_2 is equal to the expected type declared by e_1 .

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : \Pi x : A. B}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

 Dependency

We actually copy a sub-expression into the type.

This is “dependency”.

All the hard cases of type preservation have to do with these rules.

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

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

Dependency



And this happens everywhere.

For example, we also have dependent pairs

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

e is pair of an A and a B

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

This is like a cons pair, a pair of an A and a B .

Dependent Types

typing rules

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

e is pair of an A and a B

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

where

B (a type) can refer to

x (a term var. standing for $(\text{fst } e)$)

But, the type of the second component, the type of the cdr, can refer to the first component by the name x

So when we take the second component out, the type of actually the type B with the term first of e replacing x .

Dependent Types

equivalence

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$
$$P(\text{true}) \stackrel{?}{\equiv} P(\text{not false})$$

Let's talk equivalence.

In the application rule, we need to know the argument has the same type as the function's declared argument type.

But now, if terms appear in types, type equivalence can't just be syntactic identity.

For example, for some arbitrary type P , we want P of true to be equivalent to P of not false.

So we need an equivalence that can decide that during type checking.

Dependent Types

equivalence

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

$$\Gamma \vdash e_1 e_2 : B[e_2/x]$$

$$P(\text{true}) \stackrel{?}{\equiv} P(\text{not false})$$

$$\frac{A \triangleright^* v_1 \quad A' \triangleright^* v_1}{A \equiv A'}$$

Typically, this is beta/eta equivalence. Loosely, run things to values, and if the values are the same, then the types are the same.

This is why most dependently typed languages are strongly terminating and effect free.

Dependent Types

types of types

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$

$$\frac{??}{\Gamma \vdash \text{bool} : ???}$$
$$\frac{\Gamma, x : A \vdash B : ??}{\Gamma \vdash \Pi x : A. B : ???}$$

Okay so the types of terms change, and we run terms during type checking..

But types are also terms... so what are the types of types?

Like, what is the type of bool, or how do we check a Pi type?

Dependent Types

types of types: Type

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \dots$



Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \text{Type} \mid \dots$

Easy, we add a type, Type, it's the type of types.

Dependent Types

types of types: Type

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \text{Type} \mid \dots$

$\frac{}{\Gamma \vdash \text{bool} : \text{Type}}$	$\frac{\Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi x : A. e : \text{Type}}$
--	---

Then the type of bool is Type, and the type of Pi is type when B has type Type.

But wait... we've just added a term.

Dependent Types

types of types: Type

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
 $\mid \text{bool} \mid \Pi x : A. B \mid \text{Type} \mid \dots$

 $\Gamma \vdash \text{Type} : ???$

What's the type of Type? Surely, it's not Type.

Dependent Types

types of types: Type

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
| $\text{bool} \mid \Pi x : A. B \mid \text{Type}_0 \mid \text{Type}_1$
| $\text{Type}_2 \mid \text{Type}_3 \mid \text{Type}_4 \mid \text{Type}_5 \mid \text{Ty}$

$$\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}$$

Well. We'll just add, like, a lot of types.

So each type has type Type, and Type's Type is the next Type.

Dependent Types

types of types: Type

Expressions $e, A, B ::= x \mid \text{true} \mid \text{false} \mid \lambda x. e \mid e_1 e_2$
| $\text{bool} \mid \Pi x : A. B \mid \text{Type}_0 \mid \text{Type}_1$
| $\text{Type}_2 \mid \text{Type}_3 \mid \text{Type}_4 \mid \text{Type}_5 \mid \text{Ty}$

$$\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}$$

Type Theory!

Now you know why it's called type theory.

Roadmap

How do we model machine computations?

1. Dependent types
 2. Sequencing Computations
 3. Relocating Computations
 4. Lessons
- Dependency*—key to source reasoning

$$\frac{}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}} \quad \frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Higher universes *Dependency*

Okay so that's the core of dependent types.

Everything is both a term and a type, so types have types.

We have non trivial type equivalence, which evaluates terms in types.

And all our computations, like application and projection, feature dependency—copying sub-expressions into types.

Sequencing Computations

(18-20 min)

(13 min on second run)

So let's talk about control flow.

Machine don't like nested expressions. They want to do things step by step.

So our compiler needs to unnest everything and make sequencing explicit

Type Preservation 101

From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER

Cornell University

KARL CRARY

Carnegie Mellon University

and

NEAL GLEW

Cornell University

“a sequence of type-preserving transformations,
including CPS and closure conversion...”

language abstractions, such as closures, tuples, and user-defined abstract data type
system ensures that well-typed programs cannot violate these abstractions. In addition,
ing constructs admit many low-level compiler optimizations. Our translation to TALL
as a sequence of type-preserving transformations, including CPS and closure conversion.

Which for the past 20 years of type preservation research, has meant CPS
it's the canonical first pass, the canonical way to sequence computations
to make control flow explicit.

CPS

So let's take a look at CPS

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

Translate `snd e : B[fst e/x]`

into (roughly) `e+ ( $\lambda y$ . k (snd y))`

- Well, for CPS translation, encodes continuations as functions

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

Translate $\text{snd } e : B[\text{fst } e/x]$

into (roughly) $e^+ (\lambda y. k (\text{snd } y))$

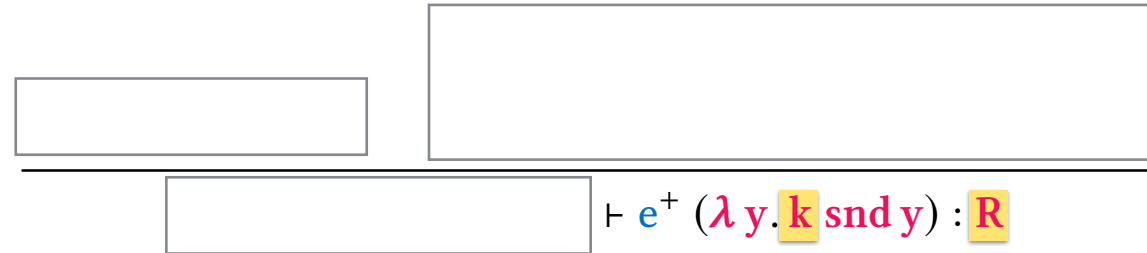
Theorem. (Type Preservation)

If $e : A$
then $e^+ : A^+$ translates to

- And our goal is to prove this: type preservation.
- if e has type A in the source, then e^+ , the translation of e , has type A^+ , the translation of A

Translate

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



- In CPS, everything produces some result type R , that I don't care about
- and has some current continuation k
-

Translate

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

$k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R}$

- The type of k in this case will be the translation of the original expression's type

-

Translate

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

$\vdash e^+ : \text{Cont} \rightarrow R$

$k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R$

- to Type check this, and application, we type check the function, e^+ which expects a continuation and produces a result.
- Here, I'm also ignoring the details of the type of continuations.
-

Translate

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

$$\frac{\vdash e^+ : \text{Cont} \rightarrow R \quad \boxed{\text{ } } \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}$$

- Then the argument, the continuation

-

Translate

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

$$\frac{\frac{}{\vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}$$

- If it's a valid continuation, then this application will succeed, and produce a result type R

-

Translate

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

$$\frac{\vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}{k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R}}$$

- So how do we type check a continuation?
- Well we encode continuations using functions.

Translate

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

$$\frac{\vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \quad \frac{k, y \boxed{} \vdash \text{snd } y \boxed{}}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R}}$$

- So we type check the body, with the variable y in scope.

Translate

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

$$\frac{\begin{array}{c} \vdash e^+ : \text{Cont} \rightarrow R \\ \hline \end{array} \quad \frac{k, y \boxed{\phantom{B[\text{fst } e/x]}} \vdash \text{snd } y \boxed{\phantom{B[\text{fst } e/x]}}}{k \vdash \lambda y. k(\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}$$

- The body is a function, k , applied to the the argument, $\text{snd } y$.
- So it suffices to check the argument has this type.
- So... does it?

Translate

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

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


$$\frac{\vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \quad \frac{k, y \boxed{} \vdash \text{snd } y \boxed{}}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R}}$$


- Well here's a hint: the typing rule for second projection from earlier.

Translate

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

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


$$\frac{\vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \quad \frac{k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R}}$$


- So we know y is a pair, and the result type has first of type for x .

Translate

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

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


$$\frac{\begin{array}{c} \vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \\ \hline \end{array} \quad \frac{\begin{array}{c} k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x] \\ \hline \end{array} \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}{\begin{array}{c} k : (B[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \text{ snd } y) : \mathbf{R} \end{array}}$$


- But this is a problem.
- These types definitely aren't equal.

Translate

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

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

$$\frac{\frac{k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}} \quad \vdash e^+ : \text{Cont} \rightarrow R}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}$$

- What we need to show is that, in this particular context, this first projection of y is equal to the translation of first of e.

Translate

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

$$(B[e_2/x])^+ \equiv B^+[e_2^+/x]$$

Need:

$$B^+[\text{fst } y/x] \equiv B^+[(\text{fst } e)^+/x]$$

$$\frac{\frac{\text{fst } e^+ : \text{Cont} \rightarrow R \quad k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}$$

- And if you look carefully, we need some other lemmas, like this substitution property.
- Staging these lemmas correctly can be hard, and I'll talk about it a little at the end of the talk.
- But I'm not going to dwell much on this...

$$(B[e_2/x])^+ \equiv B^+[e_2^+/x]$$

Need:



- YET

Translate

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

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

$$\begin{array}{c}
 \frac{\text{fst } e : \text{Cont} \rightarrow R \quad \frac{k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}{k \vdash \lambda y. k (\text{snd } y) : \text{Cont}}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R}
 \end{array}$$

- Today I'm going to focus on this: dependency.
- And this transformation messed up a dependency.
- It forgot that y only ever takes on the value of e+.

Translate $\boxed{\text{snd } e : B[\text{fst } e/x]}$ e is specific

into (roughly) $\boxed{e^+ (\lambda y. k (\text{snd } y))}$ y is arbitrary

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

$$\begin{array}{c}
 \text{snd } y : B^+[\text{fst } y/x] \\
 \hline
 \text{snd } y : B^+[\text{fst } y/x] \\
 \hline
 \text{fst } e^+ : \text{Cont} \rightarrow R \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont} \\
 \hline
 k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R
 \end{array}$$

- It forgot that y only ever takes on the value of e^+ .

Translate

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

e is specific

into (roughly)

~~$e^+ (\lambda y. k (\text{snd } y))$~~ y is arbitrary

$e^+ @ (\lambda y. k (\text{snd } y))$ y is specific

I hereby declare
That was a stupid encoding!

- Well. Let's just DECLARE that the previous encoding was stupid.
- It tried to encoding a machine step using two pre-existing feature, functions and application, that behaved complete differently than machine steps.
- So let's make a *new* feature, that behaves correctly!

Translate

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

e is specific

into (roughly)

$e^+ (\lambda y. k (\text{snd } y))$

y is arbitrary

$e^+ @ (\lambda y. k (\text{snd } y))$

y is specific

Machine-step,
not application

Continuation,
not a function

- Instead of encoding CPS using applications and functions, we acknowledge that this is not a function application, it's a machine-step.
- And this is not a function, it's a continuation.
- By making these concepts distinct, we can add a typing rule that tracks dependencies on machine steps.

$$\begin{array}{c}
 \frac{\frac{}{\vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{}{k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}}{\vdash e^+ : \text{Cont} \rightarrow R \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont}} \\
 \hline
 k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R
 \end{array}$$

X

- Instead of this stupid derivation that is broken, we create a new typing rule.

$$\begin{array}{c}
\frac{}{\vdash e^+ : \mathbf{Cont} \rightarrow \mathbf{R}} \quad \frac{}{k, y : \Sigma x : A^+. B^+ \vdash \mathbf{snd } y : B^+[\mathbf{fst } y/x]} \\
\hline
\frac{}{\vdash e^+ : \mathbf{Cont} \rightarrow \mathbf{R}} \quad \frac{}{k \vdash \lambda y. k (\mathbf{snd } y) : \mathbf{Cont}} \\
\hline
k : (B[(\mathbf{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ (\lambda y. k \mathbf{snd } y) : \mathbf{R}
\end{array}$$

X

$$\begin{array}{c}
\frac{}{\Gamma \vdash e^+ : \mathbf{Cont} \rightarrow \mathbf{R}} \quad \frac{}{k, y := e^+ \vdash \mathbf{snd } y : B^+[\mathbf{fst } y/x]} \\
\hline
\frac{}{\Gamma \vdash e^+ : \mathbf{Cont} \rightarrow \mathbf{R}} \quad \frac{}{k, y := e^+ \vdash k (\mathbf{snd } y) : \mathbf{R}} \\
\hline
k : (B[(\mathbf{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ @ (\lambda y. k (\mathbf{snd } y)) : \mathbf{R}
\end{array}$$

✓

- A typing rule that pushes the dependency up the tree.
- This says that the continuation will only ever be used when y takes on the value of e+.

$$\begin{array}{c}
\frac{\frac{}{\vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{}{k, y : \Sigma x : A^+. B^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}}{\vdash e^+ : \text{Cont} \rightarrow R \quad k \vdash \lambda y. k (\text{snd } y) : \text{Cont}} \\
\hline
k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ (\lambda y. k \text{ snd } y) : R
\end{array}$$

~~X~~

$$\begin{array}{c}
\frac{\frac{}{\Gamma \vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{}{k, y := e^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}}{\Gamma \vdash e^+ : \text{Cont} \rightarrow R \quad k, y := e^+ \vdash k (\text{snd } y) : R} \\
\hline
k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k (\text{snd } y)) : R
\end{array}$$

~~X~~ ✓

- Then, we can remember that, by the time we research this continuation applied to $\text{snd } y$, this machine step has happened.

Need:

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\frac{\frac{}{\Gamma \vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{}{k, y := e^+ \vdash \text{snd } y : B^+[\text{fst } y / x]} \quad \frac{}{k, y := e^+ \vdash k (\text{snd } y) : R}}{k : (B[(\text{fst } e) / x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k (\text{snd } y)) : R}$$

- And, hopefully, this is enough to prove these two types are equivalence.

Interpreting machine steps in type equivalence

$$\mathbf{y} := \mathbf{e}^+ \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$

- Now, we have this machine step recorded, so we track dependencies
- great.
- How do we interpret it?

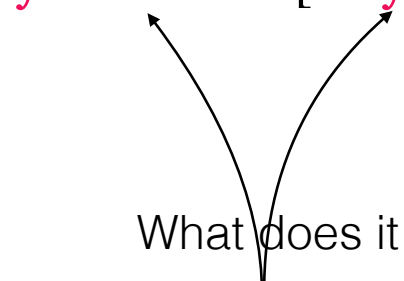
Interpreting machine steps in type equivalence

$$\mathbf{y} := \mathbf{e}^+ \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$

What does it even mean?

- What does this mean?

Interpreting machine steps in type equivalence

$$\mathbf{y} := \mathbf{e}^+ \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$


What does it even mean?

Not equality:

- $\mathbf{e}^+$ is in CPS (a function, expects a continuation)
- but $\mathbf{y}$ is a value (is a pair)

- So this “machine-step” needs an interpretation in type equivalence

Interpreting machine steps in type equivalence

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\Gamma, y := e^+ \vdash A \equiv A'$$

what continuation should we provide
to simulate the machine?

- Here's the idea: we'll just add some equivalence rules.
- To show A is equivalent to A prime under machine step y gets e^+ .
- which continuation can we provide in order to get out the value of e^+ ?

Interpreting machine steps in type equivalence

$$\mathbf{y} := \mathbf{e}^+ \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$

$$\frac{\Gamma \vdash \mathbf{A}[\mathbf{e}^+ \mathbf{id} / \mathbf{y}] \equiv \mathbf{A}'[\mathbf{e}^+ \mathbf{id} / \mathbf{y}]}{\Gamma, \mathbf{y} := \mathbf{e}^+ \vdash \mathbf{A} \equiv \mathbf{A}'}$$

The identity function
(aka halt continuation)
(aka “reset”)
(aka “runCont”)

Interpreting machine steps in type equivalence

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

Note: regular application, not “@”

$$\frac{\Gamma \vdash A[e^+ \text{id} / y] \equiv A'[e^+ \text{id} / y]}{\Gamma, y := e^+ \vdash A \equiv A'}$$

- This allows us to interpret machine-steps as function application.
- We need one more piece to finish this: we need to

Interpreting machine steps in type equivalence

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\frac{\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]}{\Gamma, y := e^+ \vdash A \equiv A'}$$

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})}$$

Type equivalence is allowed run an “un-CPS” interpreter.

- The final rule we add just interprets any machine-step as a function applied to the value of the computation e.
- This basically allows the type equivalence judgment to interpret machine steps as functions.

Proof

- Okay now we have everything we need to prove type preservation.

Proof

$$\frac{\frac{}{\Gamma \vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{k, y := e^+ \vdash \text{snd } y : B^+[\text{fst } y/x]}{k, y := e^+ \vdash k(\text{snd } y) : R}}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k(\text{snd } y)) : R}$$

- Remmber, first step was the new typing rule, which records this machine step in the environment.

Proof

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

$$\begin{array}{c}
 \frac{}{\Gamma \vdash e^+ : \text{Cont} \rightarrow R} \quad \frac{k, y := e^+ \vdash \text{snd } y : B^+[\text{fst } y / x]}{k, y := e^+ \vdash k(\text{snd } y) : R} \\
 \hline
 k : (B[(\text{fst } e) / x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k(\text{snd } y)) : R
 \end{array}$$

- Remmber, first step was the new typing rule, which records this machine step in the environment.

Proof

$$\mathbf{y} := \mathbf{e}^+ \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$

Proof

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

Rule 1

$$\frac{\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]}{\Gamma, y := e^+ \vdash A \equiv A'}$$

Rule 2

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})}$$

- Ready?
- Okay proof. Go.
- Here are the two rules.
- We have the machine-step y gets e^+ from the typing rule earlier.

Proof

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\text{By 1 } B^+[\text{fst } (e^+ \text{ id}) / x] \equiv$$

Rule 1

$$\frac{\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]}{\Gamma, y := e^+ \vdash A \equiv A'}$$

Rule 2

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})}$$

- First we apply rule 1 to the left, interpreting the machine-step as e^+ applied to the identity function.

Proof

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\text{By 1 } B^+[\text{fst } (e^+ \text{ id}) / x] \equiv$$

$$\text{By translation } \equiv B^+[e^+ @ (\lambda y. \text{fst } y) / x]$$

Rule 1

$$\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]$$

$$\Gamma, y := e^+ \vdash A \equiv A'$$

Rule 2

$$\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})$$

- Now we unroll the definition of the translation.
- the CPS translation of (fst e) says evaluation e+ with the continuation that takes the value y and does the first projection.

Proof

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\text{By 1 } B^+[\text{fst } (e^+ \text{ id}) / x] \equiv$$

$$\text{By translation } \equiv B^+[e^+ @ (\lambda y. \text{fst } y) / x]$$

$$\text{By 2 } \equiv B^+[(\lambda y. \text{fst } y) (e^+ \text{ id}) / x]$$

Rule 1

$$\frac{\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]}{\Gamma, y := e^+ \vdash A \equiv A'}$$

$$\Gamma, y := e^+ \vdash A \equiv A'$$

Rule 2

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})}$$

- Now we apply rule 2 to that machine-step; rewriting it into some function applications.

Proof

$$y := e^+ \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

$$\text{By 1 } B^+[\text{fst } (e^+ \text{ id}) / x] \equiv$$

$$\text{By translation } \equiv B^+[e^+ @ (\lambda y. \text{fst } y) / x]$$

$$\text{By 2 } \equiv B^+[(\lambda y. \text{fst } y) (e^+ \text{ id}) / x]$$

$$\text{By reduction } \equiv B^+[\text{fst } (e^+ \text{ id}) / x]$$

Rule 1

$$\Gamma \vdash A[e^+ \text{ id} / y] \equiv A'[e^+ \text{ id} / y]$$

$$\hline \Gamma, y := e^+ \vdash A \equiv A'$$

Rule 2

$$\hline \Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{ id})$$

QED

- Then apply reduction, reducing the function applications.
- And now the types are equivalent.

Summary

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{id})}$$

$$\frac{\vdash e^+ : \text{Cont} \rightarrow \mathbf{R} \quad \mathbf{k}, y := e^+ \vdash \mathbf{k} (\text{snd } y) : \mathbf{R}}{\mathbf{k} : (\mathbf{B}[(\text{fst } e)/x])^+ \rightarrow \mathbf{R} \vdash e^+ @ (\lambda y. \mathbf{k} (\text{snd } y)) : \mathbf{R}}$$

- That's the basic idea for implementing dependent machine steps with CPS.
- We don't just use functions and application, but implement a new form that treats machine steps differently, and records the dependency in the environment.
- Then we change equivalence to interpret machine steps.

Some Minor Details

Have to prove adding these is safe:

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{id})}$$

$$\frac{\vdash e^+ : \text{Cont} \rightarrow R \quad k, y := e^+ \vdash k (\text{snd } y) : R}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k (\text{snd } y)) : R}$$

There are a lot of details.

For example, we have to proof that adding these new rules is safe, and results in a well-behaved type system.

Some Minor Details

Polymorphic CPS

$e : A$

$e^+ : \Pi \alpha : \text{Type} . (A^+ \rightarrow \alpha) \rightarrow \alpha$

To prove that, we need a particular encoding of CPS, namely, polymorphic CPS.

It says that every expression e of type A is translated into, essentially,

e^+ of type for all α , $A^+ \rightarrow \alpha \rightarrow \alpha$.

Some Minor Details

Polymorphic CPS

$e : A$

$e^+ : \Pi \alpha : \text{Type} . (A^+ \rightarrow \alpha) \rightarrow \alpha$

$e^+ A^+ \text{id} : A^+$

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{id})}$$

This gives us the ability to interpret CPSd expressions as values just by applying the identity function, like we did in the new equivalence rule.

Some Minor Details

Polymorphic CPS

$e : A$

$e^+ : \Pi \alpha : \text{Type} . (A^+ \rightarrow \alpha) \rightarrow \alpha$

$e^+ A^+ \text{id} : A^+$

$$\frac{}{\Gamma \vdash e^+ @ (\lambda y. e') \equiv (\lambda y. e') (e^+ \text{id})}$$

“Continuation shuffling” (parametricity)

Formally, that equivalence rule corresponds to continuation shuffling, and is implied by parametricity, so we know its safe.

A limitation

Polymorphic CPS

$e : A$

$e^+ : \Pi \alpha : \text{Type} . (A^+ \rightarrow \alpha) \rightarrow \alpha$

$e^+ A^+ \text{id} : A^+$

$A : \text{Type}_i \rightarrow (\Pi \alpha : \text{Type}_i . (A^+ \rightarrow \alpha) \rightarrow \alpha) : \text{Type}_{i+1}$
 $(\text{Type}_i)^+ \stackrel{\text{def}}{=} \text{Type}_i$

Not type preserving with higher universes.

There are some limitations of this encoding.

This encoding doesn't work with higher universes, for onw.

The reason is essentially that the type of alpha must be at least type i, where type i is the type of A.

But this means the type of the CPS type of A is type i+1.

This breaks type preservation, since we must (for other reasons I won't go into) define the translation of Type i to be Type i.

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

Translate $\text{snd } e : B[\text{fst } e/x]$ e is specific

into (roughly) $e^+ (\lambda y. k (\text{snd } y))$ y is *arbitrary*

Let's take a step back, and look at the essence of what we've just done.

We were looking at CPS, but why?

Goal: *Type-preservation*

Translate `snd e : B[fst e/x]` e is specific

~~into (roughly) `$e^+$  ( $\lambda y. k(\text{snd } y)$ )` y is *arbitrary*~~

into (roughly) `$y := e^+; \text{snd } y$` y is *specific*

I don't care about CPS.

I care about compilation.

I care about machine steps.

Goal: *Type-preservation*

Translate $\text{snd } e : B[\text{fst } e/x]$ e is specific

into (roughly) $y := e^+; \text{snd } y$ y is *specific*

How do we encode a machine-step

and interpret type equivalence

$$\Gamma, y := e^+ \vdash B^+[\text{fst } y/x] \equiv B^+[(\text{fst } e)^+/x]$$

- Really, the question is how do we effectively encode machine steps.
- CPS had problems we were able to overcome, but maybe the **real** problem was the encoding with lambdas.

The Essence of Compiling with Continuations

Let's get to the *essence* of what we were doing compiling continuations.

ANF

(About 30 min)

Let's talk about ANF

Goal: *Type-preserving*

Translate `snd e : B[fst e/x]` e is specific

into (roughly) `y := e+; snd y` y is *specific*

- Our goal is to translation each compound expression into a machine step and a simple xpression.

Goal: *Type-preserving* **ANF** translation

Translate `snd e : B[fst e/x]` e is specific

into (roughly) `y := e+; snd y` y is *specific*

or just `let y = e+ in snd y` y is *specific*

- So why don't we just like... write that down?
- Let $y = e^+$ in $\text{snd } y$.
- Now y is specific, because we didn't FOOLISHLY write down a lambda.
- And that's basically what ANF does.

Goal: *Type-preserving* **ANF** translation

Translate `snd e : B[fst e/x]` e is specific

or just `let y =  $e^+$  in snd y` y is *specific*

well really `(let y = [·] in snd y)⟨⟨ $e^+$ ⟩⟩`

- Okay well really that would be monadic form; ANF wants to un-nest all the let bindings.
- I'll write this un-nesting operation like this: a single let bind that has a hole in it, composed (using funny angle brackets) with the translation of e^+

Goal: *Type-preserving* **ANF** translation

Translate `snd e : B[fst e/x]`

or just `let y = e+ in snd y`

well really `(let y = [·] in snd y)⟨⟨e+⟩⟩`

where `e+ = let y1 = N1 in ... let yn = Nn in N`

- We know that `e+` will be a series of let bindings follow by some body `N` that we care about

Goal: *Type-preserving* **ANF** translation

Translate

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

or just

$\text{let } y = e^+ \text{ in snd } y$

well really

$(\text{let } y = [\cdot] \text{ in snd } y) \langle\langle e^+ \rangle\rangle$

$\text{let } y_1 = N_1 \text{ in } \dots \text{let } y_n = N_n \text{ in let } y = N \text{ in snd } y$

where $e^+ = \text{let } y_1 = N_1 \text{ in } \dots \text{let } y_n = N_n \text{ in } N$

- Then the angle bracket composition will move all the let bindings out

Goal: *Type-preserving* **ANF** translation

Translate `snd e : B[fst e/x]`

or just `let y = e+ in snd y`

well really `(let y = [·] in snd y)⟨⟨e+⟩⟩`

`let y1 = N1 in ...let yn = Nn in let y = N in snd y`

where `e+ = let y1 = N1 in ...let yn = Nn in N`

- And bind N in the hole

Goal: *Type-preserving* **ANF** translation

Translate `snd e : B[fst e/x]`

to just `(let y = [·] in snd y)⟨⟨e+⟩⟩`

- But I'm just going to write it like this because all those ellipses are tedious.
- So now, we need a typing rule for this machine step.

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- Okay the typing rule for let looks approximately like this.

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- Looks very much like a typing rule for let.
- We first check that the computation e^+ , the let bound expression, is well-typed.

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- Then we check the body, with the variable in the environment.

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- The type we want is just the translation of the original type, which by our substitution lemma is equal to B^+ with first of e^+ for x .

Translate

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

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- Of course we know these aren't quite equal.
- Since the typing rule for projection copies y into the type, and we don't know that y is equal to e^+ .

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

?

- But... it's pretty close. I mean, couldn't we just have a dependent let rule and copy e^+ into the type?

Translate

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

No, because hidden ellipses

- Well you're close.
- But not quite. because these angle brackets are hiding a lot of ellipses...

Translate

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

$$\begin{array}{c}
 y \vdash \text{snd } y : B^+[\text{fst } y/x] \\
 \hline
 f : B^+[(\text{fst } e)^+/x] \rightarrow R, y \vdash f(\text{snd } y) : B' \\
 \hline
 \vdots \\
 \hline
 \vdash \text{let } y_1 = N_1 \text{ in } \dots \text{let } y_n = N_n \text{ in } \dots : B'
 \end{array}$$


because hidden ellipses = tall derivation tree

So the tree actually looks a little more like this.

Translate

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

$$\frac{\frac{y \vdash \text{snd } y : B^+[\text{fst } y/x]}{f : B^+[(\text{fst } e)^+/x] \rightarrow R, y \vdash f(\text{snd } y) : B'} \vdots}{\vdash \text{let } y_1 = N_1 \text{ in } \dots \text{let } y_n = N_n \text{ in } \dots : B'[(\text{fst } e^+)/x]}$$


And the dependency won't get resolved until later.

Translate

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

Push the dependency up the tree!

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- So we still need to push the dependency up the tree.

Translate

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

Push the dependency up the tree!

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma \mathbf{x} : A^+. B^+ \quad \mathbf{y} := \langle\langle e^+ \rangle\rangle \vdash \text{snd } \mathbf{y} : B^+[\text{fst } \mathbf{y}/\mathbf{x}]}{\vdash (\text{let } \mathbf{y} = [\cdot] \text{ in } \text{snd } \mathbf{y}) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/\mathbf{x}]}$$

- But the solution is easy, unlike in CPS.
- We don't have to reinterpret function and application, we just remember the binding introduced by let.
- Unlike in CPS, we don't need to add this rule; we can actually derive it.

Formally, dependent **let** with definitions

$$\frac{\Gamma \vdash e : A \quad \Gamma, \mathbf{x} = e \vdash e' : B}{\Gamma \vdash \mathbf{let\,x = e\,in\,e'} : B[e/x]}$$


Consistent with any type theory
(any PTS)

It's derived from this standard rule and the standard type equivalence.

This rule is an extension of dependent let with definitions.

It's been shown to be consistent with any type theory, unlike the CPS rule which required parametricity, and is already included in Coq.

Translate

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

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

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y := \langle\langle e^+ \rangle\rangle \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

- So we have a type rule, we need to show this equivalence to finish type preservation...

Interpreting machine steps in type equivalence

$$\mathbf{y} := \langle\langle \mathbf{e}^+ \rangle\rangle \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] \equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}]$$

- Okay so now what rules do we need to prove this equivalence?
- Well, I have them all listed on the next slide.
- Are you ready?

New Equivalence Rules

These are all the extra rules we need to add.

There are 0 of them.

New Equivalence Lemmas

Lemma $y := \llbracket e^+ \rrbracket \vdash y \equiv e^+$

Under machine step

$y := \llbracket e^+ \rrbracket$

y is equivalent to e^+ , interpreted as an expression

Instead, we can derive

under the machine-step y gets the value of computing e^+ , y is equivalence to e^+ as an expression

New Equivalence Lemmas

Lemma $y := \langle\langle e^+ \rangle\rangle \vdash y \equiv e^+$

$y_1 = N_1, \dots, y_n = N_n, y = N \vdash y \equiv e^+$

where $(y := \langle\langle e^+ \rangle\rangle) \stackrel{\text{def}}{=} (y_1 = N_1, \dots, y_n = N_n, y = N)$

Formally, the machine-step is encoded as a bunch of equivalences.

And we derive that under all these equivalence, y is equal to the e^+ interpreted as an expression.

Because let bindings can be interpreted either as expressions or machine-steps, this is simple.

Proof

$$y := \langle\langle e^+ \rangle\rangle \vdash B^+[\text{fst } y / x] \equiv B^+[(\text{fst } e)^+ / x]$$

- So let's do the proof.

Proof

$$\begin{aligned} y := \langle\langle e^+ \rangle\rangle \vdash B^+[\text{fst } y / x] &\equiv B^+[(\text{fst } e)^+ / x] \\ &\equiv B^+[(\text{let } y = [\cdot] \text{ in fst } y) \langle\langle e^+ \rangle\rangle / x] \end{aligned}$$

- First, unroll the translation
- It produced a let funny composed with the translation of e^+

Proof

$$\begin{aligned} y := \langle\langle e^+ \rangle\rangle \vdash B^+[\text{fst } y / x] &\equiv B^+[(\text{fst } e)^+ / x] \\ &\equiv B^+[(\text{let } y = [\cdot] \text{ in fst } y) \langle\langle e^+ \rangle\rangle / x] \end{aligned}$$

- Since we already have this exact machine-step in the context, we can simplify this term

Proof

$$\begin{aligned} \mathbf{y} := \langle\langle \mathbf{e}^+ \rangle\rangle \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] &\equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}] \\ &\equiv \mathbf{B}^+[(\mathbf{let} \mathbf{y} = [\cdot] \mathbf{in} \mathbf{fst} \mathbf{y}) \langle\langle \mathbf{e}^+ \rangle\rangle / \mathbf{x}] \end{aligned}$$

Lemma $\mathbf{y} := \langle\langle \mathbf{e}^+ \rangle\rangle \vdash \mathbf{y} \equiv \mathbf{e}^+$

- Composing $\mathbf{y}$ with this computation for $\mathbf{e}^+$ means the same thing as the machine step we have in the context, so we can inline $\mathbf{e}^+$

Proof

$$\begin{aligned} \mathbf{y} := \langle\langle \mathbf{e}^+ \rangle\rangle \vdash \mathbf{B}^+[\mathbf{fst} \mathbf{y} / \mathbf{x}] &\equiv \mathbf{B}^+[(\mathbf{fst} \mathbf{e})^+ / \mathbf{x}] \\ &\equiv \mathbf{B}^+[(\mathbf{let} \mathbf{y} = [\cdot] \mathbf{in} \mathbf{fst} \mathbf{y}) \langle\langle \mathbf{e}^+ \rangle\rangle / \mathbf{x}] \\ &\equiv \mathbf{B}^+[\mathbf{fst} \mathbf{e}^+ / \mathbf{y}] \end{aligned}$$

Lemma $\mathbf{y} := \langle\langle \mathbf{e}^+ \rangle\rangle \vdash \mathbf{y} \equiv \mathbf{e}^+$

- So now we have B+ with first of e+ for y.

Proof

$$\begin{aligned} y := \langle\langle e^+ \rangle\rangle \vdash B^+[\text{fst } y / x] &\equiv B^+[(\text{fst } e)^+ / x] \\ &\equiv B^+[(\text{let } y = [\cdot] \text{ in } \text{fst } y) \langle\langle e^+ \rangle\rangle / x] \\ &\equiv B^+[\text{fst } e^+ / y] \\ &\equiv B^+[\text{fst } y / x] \end{aligned}$$

Lemma $y := \langle\langle e^+ \rangle\rangle \vdash y \equiv e^+$

QED

- Which of course means the same thing, given this equivalence, as B^+ with first of y for x .

Summary

This slide intentionally left not-quite blank.

Here are all the non-standard rules required to do the proof.

Here are all the limitations imposed by these rules.

Of course, I have to prove these rules are safe in a dependent type theory, which I do.

The proof is trivial.

Actually Summary

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma \mathbf{x} : A^+ . B^+ \quad \mathbf{y} := \langle\langle e^+ \rangle\rangle \vdash \mathbf{snd} \mathbf{y} : B^+ [\mathbf{fst} \mathbf{y} / \mathbf{x}]}{\vdash (\mathbf{let} \mathbf{y} = [\cdot] \mathbf{in} \mathbf{snd} \mathbf{y}) \langle\langle e^+ \rangle\rangle : B^+ [(\mathbf{fst} e)^+ / \mathbf{x}]}$$

Okay really the key is being able to interpret this rule, where we remember and machine-steps and can use them to decide type equivalence.

Actually Summary

Meta-theoretic rule

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y := \langle\langle e^+ \rangle\rangle \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in } \text{snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma, x = e \vdash e' : B}{\Gamma \vdash \text{let } x = e \text{ in } e' : B[e/x]}$$

Theoretic rule

In ANF, this is implied by just remembering let bound definitions.

But we can define a nice meta-theoretic abstraction to reason about machine-steps more easily.

Actually Summary

$$\frac{\vdash \langle\langle e^+ \rangle\rangle : \Sigma x : A^+. B^+ \quad y := \langle\langle e^+ \rangle\rangle \vdash \text{snd } y : B^+[\text{fst } y/x]}{\vdash (\text{let } y = [\cdot] \text{ in snd } y) \langle\langle e^+ \rangle\rangle : B^+[(\text{fst } e)^+/x]}$$

$$\frac{\Gamma \vdash e^+ : \text{Cont} \rightarrow R \quad k, y := e^+ \vdash k (\text{snd } y) : R}{k : (B[(\text{fst } e)/x])^+ \rightarrow R \vdash e^+ @ (\lambda y. k (\text{snd } y)) : R}$$

1. No continuation type
2. **let**, unlike CPS, trivially interpreted as expression

If we compare this to the CPS rule, its very similar but with a few key differences.

First, no continuation type, which causes a problem with higher universes.

Second, let has a simple interpretation as a expression, unlike CPSed computation;

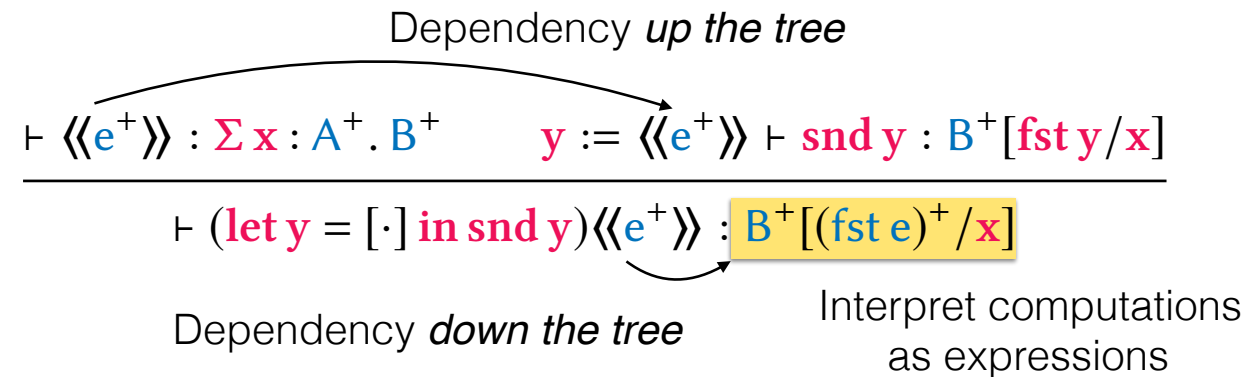
This is seen in the type of the expression, which is either still related to the original type, or a mysterious result type.

Roadmap

How do we model machine computations?

1. Dependent types
2. Sequencing Computations
3. Relocating Computations
4. Lessons

Study CPS & dependency



The lesson? The lesson is when we start unnesting expressions, we need to push dependencies up the derivation tree, in addition to pushing dependencies down the tree.

Do this by recording machine steps in the environment.

When deciding type equivalence, we need to interpret those machine steps, so we need to interpret computations as expressions

ANF gives us a nicer way of encoding that than CPS does.

Relocating Computations

(36 min?)

So computations are important.

And there's one other significant operation that compilers do with computations.

They move computations around.

Might take a function, lift it, allocate it in the heap.

Might take the branches of an if expression and lift it out into a join point.

Type Preservation 101

From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER

Cornell University

KARL CRARY

Carnegie Mellon University

and

NEAL GLEW

Cornell University

“a sequence of type-preserving transformations,
including CPS and closure conversion...”

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, the constructs admit many low-level compiler optimizations. Our translation to TAL as a sequence of type-preserving transformations, including CPS and closure conversion.

And this is captured by closure conversion.

So let's take a look.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket : \llbracket B \rrbracket$$

Usually this is explained in terms of functions, a very familiar form of delayed computation that compilers need to relocate.

I'm not going to do that.

Here's a term that I want to be relocated.

Like previous section, writing computations with angle brackets.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \langle\langle e \rangle\rangle : \langle\langle B \rangle\rangle$$

$$\Gamma \vdash \lambda z. e' : \prod z : A. B'$$

E.g., Function to be heap allocated

Maybe it's a function and I'm going to heap allocate

A *delayed computation* we want to relocate.

$$\Gamma \vdash \langle\!\langle e \rangle\!\rangle : \langle\!\langle B \rangle\!\rangle$$

To avoid too many extra greek letters, I'm just going to say it's some kind of delayed computation.

And its type some encodes the fact that it is delayed.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket : \llbracket B \rrbracket$$


So this computation may have free variables in gamma; otherwise it would be easy to move.

But remember, types also refer to terms, including free variables.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \langle\langle e \rangle\rangle : \langle\langle B \rangle\rangle$$

But this is a dependently typed language.

B can have free variables too.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

To make that clear, I'm going to make a little annotation next to the computation and the type.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash \lambda n. e^+ \boxed{} : \Pi n : \Gamma^+. B^+ \boxed{}$$

So here's how we compile this to make it relocated about.

First, we create closed term can be moved around, maybe heap allocate it.

And now to force the computation, the user just provide all of the free variables and jumps here.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash \lambda n. e^+[\text{prj}_x n] : \Pi n : \Gamma^+. B^+[\text{prj}_x n]$$

Of course, now we have to project the variable out of the explicitly passed environment n .

This has to happen both in the term and the type.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash \boxed{l :=} \lambda n. e^+[\text{prj}_x n] : \Pi n : \Gamma^+. B^+[\text{prj}_x n]$$

So first I'm going to name this computation l.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

Maybe inject a intuition here, show some assembly

$$\cdot \vdash l := \lambda n. e^+[\text{prj}_x n] \quad \Gamma^+ \vdash B^+[\text{prj}_x n]$$

Step 2: Instantiate computation

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \boxed{\phantom{\text{type}}}$$

Then I'm going to instantiate this computation.

so where it was previously used, we'll pair l with the specific variables it had before, namely a record composed of the domain of gamma.

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash \mathbf{l} := \lambda n. e^+[\text{prj}_x n] : \Pi n : \Gamma^+. B^+[\text{prj}_x n]$$

Step 2: Instantiate computation

$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle : \boxed{\phantom{\text{type}}} \times \boxed{\phantom{\text{type}}}$$

The type of this will be a PAIR, so it's still delayed

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash l := \lambda n. e^+[\text{prj}_x n] : \Pi n : \Gamma^+. B^+[\text{prj}_x n]$$

Step 2: Instantiate computation

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : (\Pi n : \Gamma^+. B^+[\text{prj}_x n]) \times \square$$

The type of this will be a PAIR, so it's still delayed

The first component with the type of l

A *delayed computation* we want to relocate.

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

Step 1: Close computation

$$\cdot \vdash \mathbf{l} := \lambda n. e^+[\text{prj}_x n] : \Pi n : \Gamma^+. B^+[\text{prj}_x n]$$

Step 2: Instantiate computation

$$\boxed{\Gamma^+} \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle : (\Pi n : \Gamma^+. B^+[\text{prj}_x n]) \times \boxed{\Gamma^+}$$

And the second component will be some record type derived from gamma.

So, if we just define $\llbracket B \rrbracket[x]^+ \stackrel{\text{def}}{=} (\prod n : \Gamma^+. B^+[\text{prj}_x n]) \times \Gamma^+$

QED?

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$



$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle : (\prod n : \Gamma^+. B^+[\text{prj}_x n]) \times \Gamma^+$$

And the second component will be some record type derived from gamma.

$$(\mathbf{B}[e_2/x])^+ \equiv \mathbf{B}^+[e_2^+/x]$$

~~QED?~~

Need:

- NOPE
- Remember this lemma? Well basically it fails, because the translation of B will be different, will have a different environment, depending on when substitution happens.

Failure of compositionality

Well-known problem:

But that's the super formal view of what happens.

But basically, it's a well-known problem with type-preserving closure conversion.

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$

Suppose we have two computations, with the same type but

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$

Suppose we have two computations, with the same type but

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$
$$\Gamma' \vdash f : \langle\langle \text{bool} \rangle\rangle \rightarrow R$$

This is particularly a problem if we ever want to use these computations in the same way. like by passing them to a function.

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$

$$(\Gamma, y)^+ \vdash \langle\langle y \rangle\rangle^+ : \Pi n : (\Gamma, y)^+. \text{bool} \times (\Gamma, y^+)$$

$$\Gamma^+ \vdash \langle\langle \text{true} \rangle\rangle^+ : \Pi n : (\Gamma^+). \text{bool} \times \Gamma^+$$

With our current translation... we get terms of different types.

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$

Well-known solution:

$$\langle\langle B[x] \rangle\rangle \stackrel{\text{def}}{=} \exists \alpha. (\Pi n : \alpha. B^+[\text{prj}_x n]) \times \alpha$$

But there's a well-known solution: using an existential type.

You existentially quantify over the type of the environment, and hide it in the type.

Failure of compositionality

Well-known problem:

$$\Gamma, y \vdash \langle\langle y \rangle\rangle : \langle\langle \text{bool} \rangle\rangle \qquad \Gamma \vdash \langle\langle \text{true} \rangle\rangle : \langle\langle \text{bool} \rangle\rangle$$

Well-known solution:

$$\langle\langle B[x] \rangle\rangle \stackrel{\text{def}}{=} \exists \alpha. (\Pi n : \alpha. B^+[\text{prj}_x n]) \times \alpha$$

$$(\Gamma, y)^+ \vdash \langle\langle y \rangle\rangle^+ : \exists \alpha. (\Pi n : \alpha. \text{bool}) \times \alpha$$

$$\Gamma^+ \vdash \langle\langle \text{true} \rangle\rangle^+ : \exists \alpha. (\Pi n : \alpha. \text{bool}) \times \alpha$$

But there's a well-known solution: using an existential type.

You existentially quantify over the type of the environment, and hide it in the type.

Failure of compositionality

and

Well-known solution:

$$\llbracket B[x] \rrbracket \stackrel{\text{def}}{=} \exists \alpha. (\Pi n : \alpha. B^+[\text{prj}_x n]) \times \alpha$$

But it doesn't work, because dependent types.

Failure of compositionality

and

Well-known solution:

$$\llbracket B[x] \rrbracket \stackrel{\text{def}}{=} \exists \alpha. (\Pi n : \alpha. B^+[\text{prj}_x n]) \times \alpha$$

Dependency says

Must project from environment **n**

In particular here.

Because of dependent types, we must project from the environment.

Failure of compositionality

and

Well-known solution:

$$\llbracket B[x] \rrbracket \stackrel{\text{def}}{=} \exists \alpha. (\Pi n : \alpha. B^+[\text{prj}_x n]) \times \alpha$$

Dependency says

Must project from environment **n**

Parametricity says

Can't project from **n**

(must hide type of **n**)

But because of the compositionality problem, we must hide the type of n

And thus can't project from it, since projection is only defined on records not on alphas

Two paths diverged in a **red** and **blue** wood*.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into

$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle$$

* Of derivation trees.

This is not the first time we've faced this problem.

And now we have a choice.

Two paths diverged in a **red** and **blue** wood*.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

CPS

Preserve the
encoding!

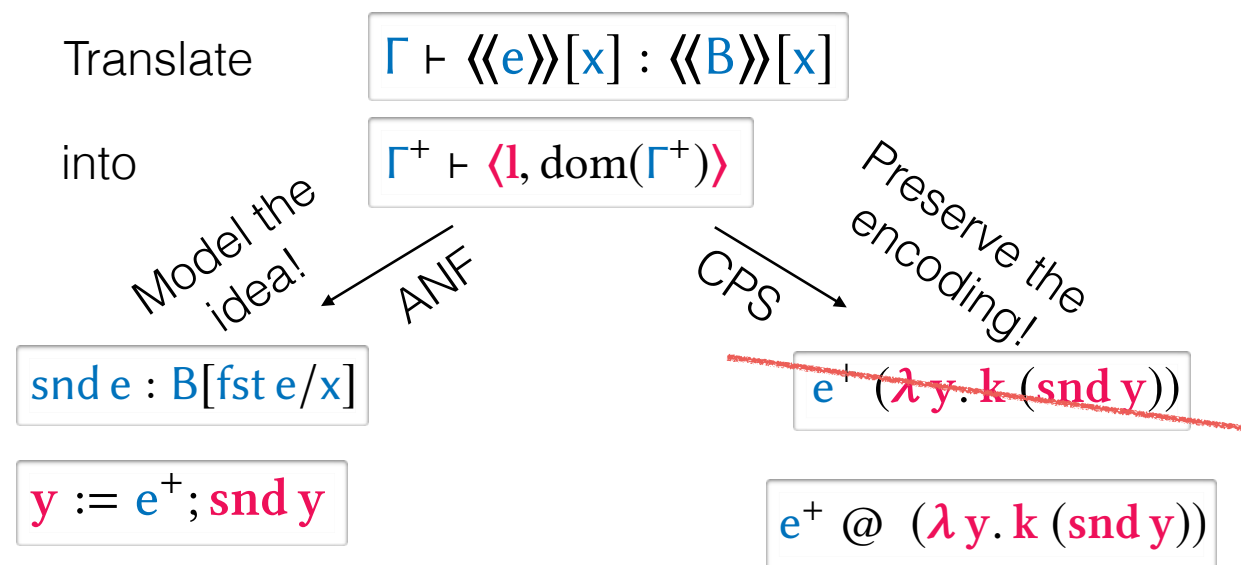
~~$$e^+ (\lambda y. k (\text{snd } y))$$~~

$$e^+ @ (\lambda y. k (\text{snd } y))$$

* Of derivation trees.

Either, we can try hard to preserve the encoding, as we did with CPS.

Two paths diverged in a **red** and **blue** wood*.



* Of derivation trees.

Or, we can try harder to model the idea of a closure in the target language.

Two paths diverged in a **red** and **blue** wood

Preserve the
encoding!
CPS

$A : \text{Type}_i \rightarrow (\prod \alpha : \text{Type}_i. (A^+ \rightarrow \alpha) \rightarrow \alpha) : \text{Type}_{i+1}$

$\langle\langle B \rangle\rangle : \text{Type}_i \rightarrow (\exists \alpha : \text{Type}_i. \dots (\prod n : \alpha. B^+) \times \alpha) : \text{Type}_{i+1}$

Not type preserving with higher universes.

If we take the road more traveled.

This one is complicated, and it's not practical—it won't scale to higher universes.

For exactly the same reason as CPS.

This existential type must have the type type 0, but the source type could have a higher universe type i.

This prevents us from showing type preservation.

Two paths diverged in a **red** and blue wood



So let's follow the road less traveled.

Translate

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

into (roughly)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

This is the translation that we want to model.

We translate a computation into a label paired with its environment.

But it's not .. literally a pair.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle \mathbf{1}, \text{dom}(\Gamma^+) \rangle$$

It's... roughly... a pair.

Because literal pairs lead to problems.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[x]$$

So let's see if we can derive the typing rule for this

The rule should look roughly like this.

Translate

$$\Gamma \vdash \llbracket e \rrbracket[x] : \llbracket B \rrbracket[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \llbracket B^+ \rrbracket[x]$$

The type is still a computation, waiting to be executed by the machine

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle$$



$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[\mathbf{x}]$$



But, there no gamma in the type, which is good

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$



$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[\mathbf{x}]$$



And the dependency structure is in-tact; the free variable x in the type B is still pointing the right place.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

$$\frac{\cdot \vdash l : (\prod n : \Gamma^+ . B^+[\text{prj}_x n])}{\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[x]}$$

What we know... is that this label l is closed.

And to pull that off, we definitely need x to be a projection from some argument, in the type.

So how do we get from there to here?

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

$$\frac{\cdot \vdash l : (\prod n : \Gamma^+ . B^+[\text{prj}_x n])}{\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[x]}$$

$$\Gamma^+ \vdash B^+[\text{prj}_x n][\text{dom}(\Gamma^+)/n] \equiv B^+[x]$$

We'll observe that if we could somehow just... replace n by the environment.

Then the result type of that label, with n replaced by the environment, would be equal to the type we want.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

$$\frac{\cdot \vdash l : (\prod n : \Gamma^+ . B^+[\text{prj}_x n])}{\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[x]}$$

$$\frac{\Gamma^+ \vdash \text{prj}_x \text{dom}(\Gamma^+) \equiv x}{\Gamma^+ \vdash B^+[\text{prj}_x n][\text{dom}(\Gamma^+)/n] \equiv B^+[x]}$$

Because, since the environment derived by gamma contains literally x,
so projecting x from that environment will just give us the right free variable.

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle$$

$$\frac{\cdot \vdash \mathbf{l} : (\prod \mathbf{n} : \Gamma^+ . B^+[\text{prj}_x \mathbf{n}])}{\Gamma^+ \vdash \langle \mathbf{l}, \text{dom}(\Gamma^+) \rangle : \langle\langle B^+ \rangle\rangle[\text{prj}_x \mathbf{n}][\text{dom}(\Gamma^+)/\mathbf{n}]}$$


$$\Gamma^+ \vdash B^+[\text{prj}_x \mathbf{n}][\text{dom}(\Gamma^+)/\mathbf{n}] \equiv B^+[\mathbf{x}]$$

Oh right, that just the dependent application rule.

Well, partial application, since this should still be a delayed computation

Translate

$$\Gamma \vdash \langle\langle e \rangle\rangle[x] : \langle\langle B \rangle\rangle[x]$$

into (*roughly*)

$$\Gamma^+ \vdash \langle l, \text{dom}(\Gamma^+) \rangle$$

or just

$$\Gamma^+ \vdash \langle\langle l \text{ dom}(\Gamma^+) \rangle\rangle : \langle\langle B^+ \rangle\rangle[x]$$

The essence of closure conversion:
delayed application

So if we model a closure as a delayed dependent application, everything, more or less, just works.

Roadmap

How do we model machine computations?

1. Dependent types
2. Sequencing Computations
3. Relocating Computations $\longrightarrow$ Closure conversion & *dependency*
4. Lessons

Model closures as delayed application; not pairs

$$\frac{\cdot \vdash \mathbf{l} : (\prod \mathbf{n} : \Gamma^+ . \mathbf{B}^+[\mathbf{prj}_x \mathbf{n}])}{\Gamma^+ \vdash \langle\langle \mathbf{l} \text{ dom}(\Gamma^+) \rangle\rangle : \langle\langle \mathbf{B}^+ \rangle\rangle[\mathbf{prj}_x \mathbf{n}][\text{dom}(\Gamma^+)/\mathbf{n}]}$$

$\equiv \langle\langle \mathbf{B}^+ \rangle\rangle[\mathbf{x}]$

Roadmap

How do we model machine computations?

1. Dependent types
 2. Sequencing Computations
 3. Relocating Computations 
 4. Lessons
- Closure conversion & *dependency*

Model closures as delayed application; not pairs

$$\frac{\cdot \vdash \mathbf{l} : (\prod \mathbf{n} : \Gamma^+ . \mathbf{B}^+[\mathbf{prj}_x \mathbf{n}])}{\Gamma^+ \vdash \langle\langle \mathbf{l} \text{ dom}(\Gamma^+) \rangle\rangle : \langle\langle \mathbf{B}^+ \rangle\rangle[\mathbf{prj}_x \mathbf{n}][\text{dom}(\Gamma^+)/\mathbf{n}]}$$


 $\equiv \langle\langle \mathbf{B}^+ \rangle\rangle[\mathbf{x}]$

Zooming out

(42 min?)

Lessons

About type preservation and dependency

What are the lessons from these two studies?

What can we learn about type preserving compilation and dependent types

Lessons

1. Compilers sequence computations

CPS

- Fails to model dependent computation sequences
- Encoding can be fixed, but not* for higher universes

ANF

- Models dependent computation sequences
- Works for higher universes

In these early stages, the compiler is doing two things.

First, sequencing computations.

CPS is the canonical way to encode this, but fails to model a dependent computation sequence, in which one computation can depend on a previous one.

Can fixed by reinterpreting the encoding, but the encoding still has problems.

ANF pretty much just works; it already models dependency sequences of computations

Lessons

1. Compilers sequence computations
2. Compilers relocate computations

CPS

- Fails to model dependent computation sequences
- Encoding can be fixed, but not* for higher universes

Parametric CC

- Fails to model dependent closures
- Encoding can be fixed, but not* for higher universes

ANF

- Models dependent computation sequences
- Works for higher universes

Abstract CC

- Models dependent closures
- Works for higher universes

The second job of the early compiler stages is to relocate computations, or at least make them relocatable.

This is what closure conversion does; a closure is just a relocatable computation.

The standard approach to type-preserving closure conversion, the parametric closure conversion, fails to model dependent closures, although it can be fixed.

But abstract closure conversion models dependent closures just fine.

Changes the types of types

Lessons

1. Compilers sequence computations
2. Compilers relocate computations

CPS

ANF

- Fails to model dependent

- Models dependent

Lesson:

- Dependent types do not suffer bad encodings gladly.

not* for higher universes

Parametric CC

Abstract CC

- Fails to model dependent closures

- Models dependent closures

- Encoding can be fixed, but not* for higher universes

- Works for higher universes

The over all lesson?

Dependent types don't let you get away with bad encodings.

The standard type preserving encodings fail to capture some invariants, and this is exposed when we try to preserve dependent types.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

I can also give a recipe for type preservation, informed by these studies.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$


Dependent-type-preservation recipe:

0. Model dependent machine computation

Step 0. model the dependency in your low-level object.

This, of course, is the hard part, and what I've focused on in this talk.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

0. Model dependent machine computation

1. Prove compositionality $(B[e/x])^+ \equiv B^+[e^+/x]$

After that, you prove compositionality.

That we can reason about the translation of a composed term by translating its parts and then composing them.

This comes from the fact that types in the source have substitutions.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

0. Model dependent machine computation

1. Prove compositionality $(B[e/x])^+ \equiv B^+[e^+/x]$

2. Prove equivalence preservation If $A \equiv A'$ then $A^+ \equiv A'^+$

Next, prove equivalence preservation.

The type system has an equivalence judgment; you must preserve it.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

0. Model dependent machine computation

1. Prove compositionality

$$(B[e/x])^+ \equiv B^+[e^+/x]$$

2. Prove equivalence preservation

$$\text{If } A \equiv A' \text{ then } A^+ \equiv A'^+$$

a. Prove reduction preservation

$$\text{If } e \mapsto e' \text{ then } e^+ \mapsto^* e'^+$$

When equivalence is defined by evaluation, which is common, you can break this into two lemmas.

First, preserve a single step of reduction 0 or more steps of reduction in the target.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

0. Model dependent machine computation

1. Prove compositionality

$$(B[e/x])^+ \equiv B^+[e^+/x]$$

2. Prove equivalence preservation

$$\text{If } A \equiv A' \text{ then } A^+ \equiv A'^+$$

a. Prove reduction preservation

$$\text{If } e \mapsto e' \text{ then } e^+ \mapsto^* e'^+$$

b. Prove conversion preservation

$$\text{If } e \mapsto^* e' \text{ then } e^+ \mapsto^* e'^+$$

Second, preserve conversion, or multiple steps of reduction under any context.

Lessons

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$

Dependent-type-preservation recipe:

0. Model dependent machine computation

1. Prove compositionality $(B[e/x])^+ \equiv B^+[e^+/x]$
2. Prove equivalence preservation If $A \equiv A'$ then $A^+ \equiv A'^+$
 - a. Prove reduction preservation If $e \mapsto e'$ then $e^+ \mapsto^* e'^+$
 - b. Prove conversion preservation If $e \mapsto^* e'$ then $e^+ \mapsto^* e'^+$
3. Prove type preservation If $\Gamma \vdash e : A$ then $\Gamma^+ \vdash e^+ : A^+$

Then, you ought to be able to prove type preservation.

Lessons

Correct separate compilation recipe:

0. Defined related observations

$\text{true} \approx \text{true}$ $\text{false} \approx \text{false}$

1. Prove compositionality

$$(B[e/x])^+ \equiv B^+[e^+/x]$$

2. Prove equivalence preservation

$$\text{If } A \equiv A' \text{ then } A^+ \equiv A'^+$$

a. Prove reduction preservation

$$\text{If } e \mapsto e' \text{ then } e^+ \mapsto^* e'^+$$

b. Prove conversion preservation

$$\text{If } e \mapsto^* e' \text{ then } e^+ \mapsto^* e'^+$$

Finally, if you follow this recipe, and you define a cross-language relation relating source and target observations.

Something like blue true is related to red true

Lessons

Correct separate compilation recipe:

0. Defined related observations

$\text{true} \approx \text{true}$ $\text{false} \approx \text{false}$

$v \approx v$ only if $v^+ \equiv v$

1. Prove compositionality

$(B[e/x])^+ \equiv B^+[e^+/x]$

2. Prove equivalence preservation

If $A \equiv A'$ then $A^+ \equiv A'^+$

a. Prove reduction preservation

If $e \mapsto e'$ then $e^+ \mapsto^* e'^+$

b. Prove conversion preservation

If $e \mapsto^* e'$ then $e^+ \mapsto^* e'^+$

Then you need to prove this is implied by the equivalence language in the target language, which ought to be trivial if you translate observations correct

Take away

$$\frac{\Gamma \vdash e_1 : \Pi x : A. B \quad \Gamma \vdash e_2 : A' \quad A \equiv A'}{\Gamma \vdash e_1 e_2 : B[e_2/x]}$$


Dependent-type-preservation recipe:

0. Model dependent machine computation

Type-preserving compilation of dependently typed languages is a viable technique for statically eliminating classes of errors introduced during compilation and linking.

There's a lot more I want to say, but I'll leave you with this 1 lesson; step 0 of the recipe.

Type preserving compilation is possible.

It requires rethinking the encodings we've been using, and instead focusing on model dependency in low-level objects like sequences of computations and closures.