Compiler Correctness

Perhaps the most famous verified compiler out there is CompCert. This project was started by Xavier Leroy in 2004-2005. It is a C compiler written entirely in Coq. It is an impressive feat: it is a realistic language that is being compiled. It has 11 (or perhaps 14?) passes which goes from C-light to three different backends, PowerPC, x86 and ARM. This whole thing has been formalized.

What I'm going to talk about today is compiler correctness theorems themselves, and what it is that we are proving about compilers.

CompCert C

<u>Thm</u> $P_S \rightsquigarrow P_T \Rightarrow P_S \simeq P_T$

Notationally, we will be using blue to represent terms in the source language, and red to represent terms in the target language.

Intuitively, when a compiler is correct, the source and target programs should be the same. So one way to express this is to specify a mathematical equality between the *denotation* of programs. We don't actually do this, because taking the denotation of programs is difficult. Instead, we define some appropriate "equivalence" relation. Now, how this is defined is very subtle, and has important implications on how you will proceed with your proof.

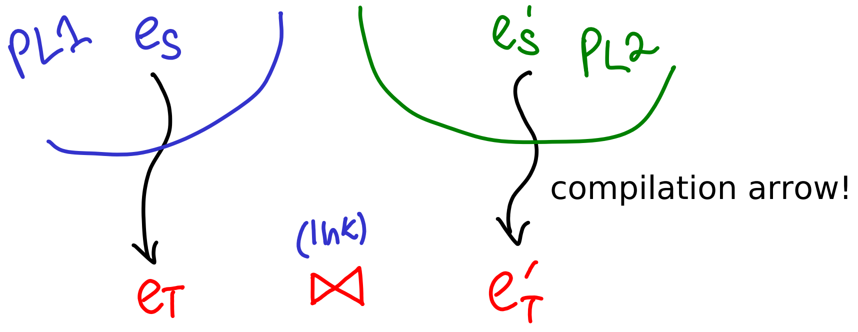
$$[\![P_S]\!] = [\![P_T]\!]?$$

$$\begin{tikzcd}
P_S \arrow[r, mapsto] \arrow[d, dash, dotted] & \cdot \arrow[r, mapsto] & P_{S_i} \arrow[r, mapsto] \arrow[d, dash, dotted] & P_{S_{i+1}} \arrow[d, dash, dotted] \\
P_T \arrow[r, mapsto] & \cdot \arrow[r, mapsto] & P_{T_i} \arrow[r, mapsto, "*"] & P_{T_{i+1}}
\end{tikzcd}$$

In CompCert, you carry out a simulation argument, where if you take a step in the source language, this correspond to zero or more steps in the target language.

There is an important caveat here, which is that CompCert's theorem only works if you compile a whole program. But we don't usually compile a whole program; we want to link against libraries.

What I'd like to do is take a source program, compile it to a target language, link it against another program (assembly, low level libraries, or even a program in another language):



As a concrete example, consider .NET: .NET is an intermediate language or platform designed by Microsoft. The idea is that there are many frontend languages which compile down to .NET IL.

$$\begin{tikzcd}
\text{F\#} \arrow[dr] & \text{C\#} \arrow[d] & \text{Visual Basic} \arrow[dl] \\
& \text{.NET IL} &
\end{tikzcd}$$

(Previously, we were considering an intermediate language which was assembly.) This is a common way people achieve language interoperability today. (Of course, you might run into trouble if you wanted to link .NET IL against assembly).

The upshot is that when we have multiple languages, they need to be compiled down some common intermediate representation, at which point they can be linked together. Once this step is achieved, when can use a CompCert style whole program proof; however, for all of the earlier steps, we will need to be able to verify the passes in absence of a full program.

Allowing linking is equivalent to allowing open terms (who are filled in with things provided by the linker. So our statement of equivalence is now over expressions, not programs.

$$e_S \rightsquigarrow e_T \Rightarrow e_S \simeq e_T$$

While we have generalized our problem to expressions, we still haven't said what it means for two partial programs in different languages to be equivalent. Today, we will look at two methods for achieving this. The first will use a logical relation to say what it means for two expressions to be equal, across the language barrier:

$$\underline{\text{Thm}} \quad \Gamma_S \vdash e_S : \tau_S \rightsquigarrow e_T \Rightarrow \Gamma_S \vdash e_S \underset{\text{logically related}}{\propto} e_T : \tau_S$$

am open program with some free variables, the slots for the things to be linked in.

this symbol is purposely asymmetric to emphasize the fact that the are different languages

So, I am going to do a simple example of a STLC with booleans, and compile into a STLC with integers and recursive types (providing non-termination).

SOURCE $\quad\quad \Gamma \vdash e : \tau \quad\quad \Gamma ::= \cdot \mid \Gamma, x:\tau$

$\tau ::= \text{bool} \mid \tau_1 \to \tau_2$

$e ::= \text{true} \mid \text{false} \mid \text{if}(e, e_1, e_2) \mid x \mid \lambda x:\tau.e \mid e_1\, e_2$

$v ::= \text{true} \mid \text{false} \mid \lambda x:\tau.e$

$E ::= [\cdot] \mid \text{if}(E, e_1, e_2) \mid E\, e \mid v\, E$

TARGET $\quad\quad \Gamma \vdash e : \tau \quad\quad \Gamma ::= \cdot \mid \Gamma, x:\tau$

$\tau ::= \text{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \mu\alpha.\tau$

$e ::= n \mid \text{if} \begin{bmatrix}\text{arith}\\ \text{cmp}\end{bmatrix} e_1\, e_2 \mid \ldots \mid x \mid \lambda x:\tau.e \mid e_1\, e_2 \mid \text{fold}\, e \mid \text{unfold}\, e$

$v ::= n \mid \lambda x:\tau.e \mid \text{fold}\, v$

$E ::= [\cdot] \mid \text{if}\, E\, e_1\, e_2 \mid E\, e \mid v\, E \mid \text{fold}\, E \mid \text{unfold}\, E$

Why non-termination in the target language? I want to make it clear that there are often many things in low-level code that you cannot write in a high level language. So the target has some feature which the source doesn't, that is non-termination.

Q: Are you going to prove a completeness theorem?
A: What is meant by "completeness"? Perhaps you are speaking of the fact that we are proving only one direction: if S evaluates to SV, then T evaluates to TV. So sometimes you need to show the other direction, though for our case it is not necessary. CompCert TSO, which deals with C with relaxed memory model and concurrency. With concurrency, you often need both backward simulation

$$\begin{tikzcd}
e_S : \tau_S \arrow[d, squiggly, "I"'] \\
e_I : (\tau_S)^I \arrow[d, squiggly, "T"'] \\
e_T : (\tau_S)^{IT}
\end{tikzcd}$$

transformation on type

We're going to write a small type-preserving compiler for our language.
What is type preserving compilation? For a long time, even when you were compiling a typed language, you would first do type checking for the source language, and if it did type check, you threw away the types, and translated the untyped syntax down the successive phases of the compiler. In the early 90s, at CMU, people like Harper and Morrisett started working on type-preserving compilation. The idea was that when you defined your language, if you had type information at the source level, that information should not be thrown away; it should be transformed from phase to phase. This would require your intermediate languages to be typed. But was really nice was it gave you a really nice way of debugging your compiler. If your compiler has a mistake in a pass and produces a target code that doesn't type check, you'd immediately know that you'd done something wrong. Even Java to Java bytecode preserves types, since the JVM bytecode has lots of security checks.

COMPILER

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau_S \rightsquigarrow x}$$

$$(\text{bool})^+ = \text{int}$$
$$(\tau_1 \to \tau_2)^+ = \tau_1^+ \longrightarrow \tau_2^+$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool} \rightsquigarrow 1} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool} \rightsquigarrow 0}$$

invariant: $$\frac{\Gamma \vdash e : \tau \rightsquigarrow e}{\Gamma^+ \vdash e : \tau^+}$$

$$\frac{\Gamma \vdash e : \text{bool} \rightsquigarrow e \qquad \Gamma \vdash e_2 : \tau \rightsquigarrow e_2}{\Gamma \vdash \text{if}(e, e_1, e_2) : \tau \rightsquigarrow \text{if}(e \geq 1, e_1, e_2)}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau' \rightsquigarrow e}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau' \rightsquigarrow \lambda x : \tau^+.\, e} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \rightsquigarrow e_1 \qquad \Gamma \vdash e_2 : \tau' \rightsquigarrow e_2}{\Gamma \vdash e_1\ e_2 : \tau \rightsquigarrow e_1\ e_2}$$

The type-directed compiler can be divided into two parts: first, we define a type translation (the superscript plus), which tells us what the types of compiled expressions should be (as expressed by the invariant). Then, our translation proceeds as per all of the typing-derivations in the language.

Now let's setup our logical relation, which says when two terms (in different languages) are equivalent (notice the relation is defined only on the source types):

$$\mathcal{V}[\![\text{bool}]\!] = \{(\text{true}, n) \mid n \geq 1\} \cup \{(\text{false}, n) \mid n < 0\}$$
$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \{(\lambda x{:}\tau_1. e_1, \lambda x{:}\tau_1^+. e) \mid \forall v_S, v_T \in \mathcal{V}[\![\tau_1]\!]. (e[v_S/x], e[v_T/x]) \in \mathcal{E}[\![\tau_2]\!]\}$$
$$\mathcal{E}[\![\tau]\!] = \{(e_S, e_T) \mid \forall v_S.\ e_S \overset{*}{\longmapsto} v_S \Rightarrow \exists v_T.\ e_T \overset{*}{\longmapsto} v_T \wedge (v_S, v_T) \in \mathcal{V}[\![\tau]\!]$$

A few notes: while our target language may be non-terminating, our source language is terminating, so we can simply just require the target to be terminating for the expressions to be related. Additionally, our transformation is quite simple, but if you were doing something such as closure conversion, your types would get a bit more complicated, since you'd have to translate certain types into existentials. However, if you follow the types, you should still be able to figure out the translation! There are no step-indexes, because we're defining the relation on the type structure of the *source* language.

$$\mathcal{G}[\![\cdot]\!] = \{(\emptyset, \emptyset)\}$$
$$\mathcal{G}[\![\Gamma, x{:}\tau]\!] = \{(\gamma_S[x \mapsto v_S], \gamma_T[x \mapsto v_T]) \mid (\gamma_S, \gamma_T) \in \mathcal{G}[\![\Gamma]\!] \wedge (v_S, v_T) \in \mathcal{V}[\![\tau]\!]\}$$

$$\Gamma_S \vdash e_S \propto e_T : \tau_S \overset{\text{def}}{=} \forall (\gamma_S, \gamma_T) \in \mathcal{G}[\![\Gamma_S]\!] \Rightarrow (\gamma_S(e_S), \gamma_T(e_T)) \in \mathcal{E}[\![\tau_S]\!]\}$$

Digression: bisimulation is abused term in the community. It refers to a specific proof technique, but it is sometimes used to describe when there is some simulation relationship between two processes. The proof technique requires coinduction. And you cannot use bisimulation to directly show contextual equivalence, since you cannot run open terms.

IMPORTANT POINT: We had to write down our logical relation in order to be even able to STATE what our compositional correctness theorem was.

We can prove our correctness theorem (semantics preserving compilation) using induction on derivations.

fundamental property

$$\underline{\text{Thm}} \quad \Gamma_S \vdash e_S : \tau_S \rightsquigarrow e_T \Rightarrow \Gamma_S \vdash e_S \propto e_T : \tau_S$$

Q: What happens if type safety doesn't hold, e.g. an untyped language?
A: Type preserving compilation doesn't really work in the way I've described. But I guess you can still set up a logical relation with step-indexes.

Digression: This is not the only theorem that one might be interested in. Consider another theorem (called "equivalence preservation"):

$$\begin{tikzcd}
\Gamma \vdash & e_{S_1} \arrow[d, squiggly] & \approx^{ctx}_{S} & \hat{e}_S \arrow[d, squiggly] & : \tau \\
\Gamma^+ \vdash & e_T & \approx^{ctx}_{T} & \hat{e}_T & : \tau^+
\end{tikzcd}$$

equiv-preserv $\Downarrow$ $\quad \Uparrow$ equiv-reflect $= $ fully abstract

This is insanely difficult to prove! The reason is this: the target language could have some pretty low-level operations to distinguish between otherwise equivalent source programs (e.g. something using a stack versus using a list). So what you would need is to show that "encapsulation" is also preserved, and you might expect this to not generally be true for most low level languages. But this is still a desirable property: this theorem would allow us to ensure that security guarantees in the source level are preserved. So we might also call this type of compilation "security preserving compilation." As an example, F* is a dependently typed extension of F#, which allows you to prove all sorts of properties of your language. You would like to make sure these invariants are preserved, and it is something that is open research.

By the way, when the implication goes in the other way, it goes equivalence reflecting, and when it goes both ways, it is fully abstract. There was a recent paper on this subject.

Returning to semantics preserving compilation, we have defined equivalence in terms of a logical relation. But what does it really mean? Take this sample program and its compilation.

$$\begin{tikzcd}
f:\mathrm{bool}\to\mathrm{bool} \vdash f\ \mathrm{false} : \mathrm{bool} \arrow[dd, squiggly] & & e_S' & \\
 & & & e_O' \arrow[dl, squiggly] \\
f:\mathrm{int}\to\mathrm{int} \vdash f\ 0 : \mathrm{int} & \bowtie & e_T' \arrow[uu, dotted] &
\end{tikzcd}$$

Even though we compiled e_t' from some other language...

Imagine $f_1 = \lambda x{:}\mathrm{int}.\ 0$ $\qquad f_1 = \lambda x{:}\mathrm{bool}.\ \mathrm{false}$

What does our compiler correctness theorem says? Well, our statement of logical relation requires the substitutions to be related. So I've given an f1, but I need to come up with an es' which related, in order to do this substitution. In this particular case, we can imagine up the right source term:

Q: This only works because you only had integers in your source language.
A: I am about to make a point like this. Before that, here is another example that works:

$f_2 = \lambda x{:}\mathrm{int}.\ \mathrm{if}\ x>0\ \mathrm{then}\ 0\ \mathrm{else}\ 1$ $\qquad f_2 = \lambda x{:}\mathrm{bool}.\ \mathrm{if}\ x\ \mathrm{then}\ \mathrm{true}\ \mathrm{else}\ \mathrm{false}$

And now here are some examples that do not work:

$f_3 = \lambda x{:}\mathrm{int}.\ x+1$
$f_4 = \lambda x{:}\mathrm{int}.\ \Omega$

We intentionally chose the target language to use ints to make this example fail; if it was nat, f3 would just be constant true.

These are really trivial languages, so the fact that we are having this difficulties elucidates what our compiler correctness theorem actually means. What we have effectively said, is that we can only link with things which *could* have been compiled from the source language, i.e. you cannot link with anything that you could not have written in the source.

The theorem is not EVEN THINKING about things that are not in the source language. And remember, that's our fault, because we wrote the definition! So next time you read a compiler correctness paper, don't just say, "Hey, they proved correctness!" You have to read the theorem, you have to understand what is going on under the covers, to see what actually happened.

To make things worse, when you add references, and you need Kripke logical relations (with things on worlds), it is really hard to wade through the logical relation to find out what the correctness theorem is really saying.

Q: Well, is this a problem if your source language is really expressive?
A: You could say with a Turing-complete language, you'd always be able to *find a way*. But you'd still run into problems with type-directedness. Imagine you were linking with something like int -> int. Surely you could imagine some function which had that type, but modified some state. You wouldn't be able to represent it with the type.

By the way, there is another weakness with the original approach.
People who work in compositional correctness talk about vertical compositionality and horizontal compositionality. Horizontal compositionality is about linking, while vertical compositionality is about multipass compilers:

$$e_S \rightsquigarrow e_I \rightsquigarrow e_T$$

You should be able to prove each pass correct independently, and then easily combine them. The problem with the previous approach, is once you scale this approach up (e.g. if you have references in your source), is that you'd need a step-indexed logical relation, and these logical relations ARE NOT transitive.
(We discussed this problem in previous lectures.)

$$e_S \propto e_I : \tau_S \qquad e_I \propto e_T : \tau_I \overset{?}{\Longrightarrow} e_S \propto e_T : \tau_S$$

Nota bene: within a single language, you have a statement:

$$\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau \wedge \Gamma \vdash e_2 \lesssim^{ctx} e_3 : \tau \Rightarrow \Gamma \vdash e_1 \lesssim^{ctx} e_3 : \tau$$

Contextual equivalence is obviously transitive. So if I am only in one language, I can do a trick where I show my logical relation is sound and complete with respect to contextual equivalence, and then I automatically get that it's transitive, I don't even need to crank the proof.

In a multi-language definition, there is no notion of contextual equivalence to appeal to. But that's exactly the idea that will underly our next notion of equivalence.

So I want to talk about a different way of doing things that allows us to link with arbitrary e_t'. So we only require that it is of the right type. It can be a simple type or a logical specification. At the assembly level, this might be a question of calling conventions. But I want to allow linking with f3 and f4.

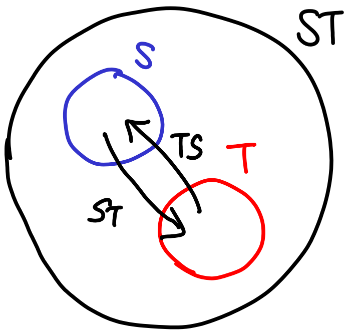
Let's have a new specification of "equivalence" between programs. Intuitively, what we want to say is:

$$\begin{tikzcd} e_S \arrow[d, squiggly] & \overset{?}{\bowtie} & e_T' \\ e_T & \bowtie & e_T' \end{tikzcd}$$

Informally, this says that we want some way of linking S things with T things, and then say linking eS and eT' is the same as linking eT and eT'. Of course, if eT' introspects on the structure of eT, then this will not work; so we need to forbid that.

Of course, there is a question mark above one of these linkings, so we have to say something about how we're going to link things which are in different languages.

Here is what we'll do: let's define a big language ST which embeds both S and T in itself:



However, S and T are not permitted to interact with each other in an unrestricted way: we will have specific boundaries between S and T that allow them to interoperate.

To define this language, we literally copy paste the old definitions, semantics and typing rules, and just add some ST/TS forms for interoperability.

SOURCE $\qquad \Gamma \vdash e : \tau \qquad \Gamma ::= \cdot \mid \Gamma, x:\tau$

$$\tau ::= \text{bool} \mid \tau_1 \to \tau_2$$
$$e ::= \text{true} \mid \text{false} \mid \text{if}(e, e_1, e_2) \mid x \mid \lambda x:\tau.e \mid e_1\, e_2 \mid {}^{\tau}\text{ST}\, e$$
$$v ::= \text{true} \mid \text{false} \mid \lambda x:\tau.e$$
$$E ::= [\cdot] \mid \text{if}(E, e_1, e_2) \mid E\, e \mid v\, E \mid {}^{\tau}\text{ST}\, E$$

always evaluate in the boundary down to a value, before translating

TARGET $\qquad \Gamma \vdash e : \tau \qquad \Gamma ::= \cdot \mid \Gamma, x:\tau$

$$\tau ::= \text{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \mu\alpha.\tau$$
$$e ::= n \mid \text{if} \begin{bmatrix}\text{arith}\\ \text{cmp}\end{bmatrix} e_1\, e_2 \mid \ldots \mid x \mid \lambda x:\tau.e \mid e_1\, e_2 \mid \text{fold}\, e \mid \text{unfold}\, e \mid \text{TS}^{\tau}\, s$$
$$v ::= n \mid \lambda x:\tau.e \mid \text{fold}\, v$$
$$E ::= [\cdot] \mid \text{if}\, E\, e_1\, e_2 \mid E\, e \mid v\, E \mid \text{fold}\, E \mid \text{unfold}\, E \mid \text{TS}^{\tau}\, E$$

COMBINED $\qquad \Gamma \vdash e : \tau \qquad \Gamma ::= \cdot \mid \Gamma, x:\tau_s \mid \Gamma, x:\tau_s$

$$\tau := \tau \mid \tau$$
$$e := e \mid e$$
$$v := v \mid v$$
$$E := E \mid S$$

$$\frac{\Gamma \vdash e : \tau^+}{\Gamma \vdash {}^{\tau}\text{ST}\, e : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{TS}^{\tau}\, e : \tau^+}$$

The most interesting thing about this combination is what happens operationally. The problem is how to bring values from the source language to the target language.

$${}^{\text{bool}}\text{ST}\, n \longmapsto \text{true} \quad \text{if } n > 0$$
$${}^{\text{bool}}\text{ST}\, n \longmapsto \text{false} \quad \text{if } n \leq 0$$
$$\text{TS}^{\text{bool}}\, \text{true} \longmapsto 1$$
$$\text{TS}^{\text{bool}}\, \text{false} \longmapsto 0 \quad (\text{but } -2 \text{ is fine!})$$

$$\underset{\tau_1^+ \to \tau_2^+}{{}^{\tau_1 \to \tau_2}\text{ST}\, v} \longmapsto \lambda x:\tau_1.\, {}^{\tau_2}\text{ST}\, (\underbrace{v\, (\underbrace{\text{TS}^{\tau_1}\, x}_{\tau_1^+})}_{\tau_2^+})$$

Exercise (trivial): do the other direction.

Notice: Post-lecture, we noticed that there was a problem with this definition, that is, it does not preserve the cancellation rules (TS (ST 3) != 3). We couldn't really think of a good fix for the problem. However, in actually developments, this tends not to come up, since in the target language we tend to have ways of expressing types which are isomorphic to the types in the source language. Beware, though!

We now have a single language for ST. And since it is a single language, we can define contextual equivalence for it. (You know how to do that). So thus, our theorem is:

$$\underline{\text{Thm}} \quad \Gamma_S \vdash e_S : \tau_S \rightsquigarrow e_T \;\Rightarrow\; \Gamma_S \vdash e_S \approx^{ctx}_{ST} \overbrace{{}^{\tau_S}ST\; \underset{\tau_S^+}{e_1}}^{\tau_S} : \tau_S$$

The beautiful thing about this is that I no longer have to deal with the messiness of the two languages; I now only have a single language.

This definition assumes a type preserving compiler (and we think this is a good restriction), because it argues that code of type bool which is allowed to link against other things, it is allowed to link against a translation of the type.
This information is important, because it tells you what is sensible to link with.
I don't think it's essential to have types at the target language, all you need is a program logic. I wish for a dependently typed assembly language which can express rich properties.

This equivalence solves the horizontal composition problem, since you are now able to link everything with everything else. Furthermore, you are able to do vertical composition:

$$e_S \approx^{ctx} SIT\; e_t \quad \begin{tikzcd} S \arrow[d, "SI \quad IS"] \\ I \arrow[d, "IT \quad TI"] \\ T \end{tikzcd}$$

$$e_S \rightsquigarrow e_I \;\Rightarrow\; e_S \approx^{ctx}_{SI} SI\; e_I$$

$$e_I \rightsquigarrow e_T \;\Rightarrow\; \underbrace{e_I \approx^{ctx}_{IT} IT\; e_T}_{SI\; e_I \approx^{ctx}_{SIT} SI(IT\; e_T)}$$

But note that we need to say that it is contextually equivalent to when something is wrapped in ST. Contextual equivalence is hard to prove, so you should use logical relations.

When does this get hard? When you have polymorphism in your source and your target language, the interoperability boundary between the two can be a bit tricky. When both languages have mutable references is also interesting: you can't just make a copy of the location to go across the boundary.

This operational semantics captures what it means for a source thing of some type to be related to a target thing of a related type. I don't think this should be characterized as compilation. The driving force is the type transformation.

An important thing you need to prove when doing contextual equivalence are these:

$$e_S \approx^{ctx} ST\ TS\ e_S : \tau_S$$
$$e_T \approx^{ctx} TS\ ST\ e_T : \tau_S^{+}$$

If this doesn't hold, then you've done something wrong. You don't have free rein over your definition of contextual equivalence.

The target language would have to have parametricity. If eS is typed, it has requirements about what it wants to be linked with. It needs to capture things that the source language cares about. So maybe this means this technique will not scale.

By the way, the original paper for language interoperability between a typed language like STLC and an untyped language was by Matthews-Findler POPL '07. In interoperability work, people have been doing FFIs and a lot of hacking, and what their work did was really say we should put this on formal ground. We also had a more recent paper on CPS transformation in ICFP'11.

Q: Haven't you told us that you have a way of combining any two programming languages together?
A: Yes, but remember, I am not allowing unrestricted combinations, I am forcing the languages to only interact through these boundaries, mediated through my type translation!