In a large-scale software system, each part of the system is responsible for a specific task and is, therefore, often written in a language that is best suited for that task. This is a problem because as programmers develop complex systems, they spend much time refactoring. (making changes to components that should result in equivalent behavior.)

Unfortunately, in a refactoring process programmers cannot solely rely on contextual equivalence of their own language. Since different languages interact after they have been compiled to a target language, programmers have to take the contexts that is inexpressible in their source language but is expressible in the common target into account in their reasoning.

For example, even if a programmer is using a safe language like OCaml, the equivalence he/she relies on in the source level can be disrupted by a C code that can be linked in a lower-level unsafe language.

We would like programmers to be able to reason using contextual equivalence in the language they used even in the presence of target-level linking. A fully abstract compiler does this. If two components are contextually equivalent at the source their compiled versions are contextually equivalent at the target.

However, this comes at a steep cost: a fully abstract compiler disallow linking with components whose behavior is inexpressible in the compiler's source language. What if this extra behavior or control is exactly what the programmer wants, but was not able to express in his/her source language? Moreover, we want programmers to decide what kind of linking is necessary in his/her program, not compiler writers.

(Similar work: cross-langauge linking is supported by Compositional Compcert, but it only allows linking with components that satisfy CompCert' memory model, Perconti and Ahmed's multi-language style of verified compilers, but a programmer needs to understand the full ST language and the compiler from R to T.)

The method advocated in the paper is to extend source language specifications with linking types. This is a better solution than previous works because we minimally enrich source language types and allow programmers to annotate only when they want to link with inexpressible components in their source language.

To introduce linking types formally, we consider two simple source languages. λ is the simply typed lambda calculus with integer base types and λ^{ref} extends λ with mutable references. We want type-preserving fully abstract compilers from λ and λ^{ref} to a common target language. The target should have a rich enough type system to allow full abstract type translation and to use types to rule out equivalence disrupting linking.

For example, our target $\lambda^{\text{ref}}_{\text{exn}}$ has a modal type system that can distinguish pure computation from impure computation, which then can be used to rule out linking \lambda with λ^{ref} component that uses mutable references. We also added exceptions to the target to represent the extra control flow commonly found in low-level language.

Let me illustrate the linking idea with e_1 and e_2 here. These two programs are equivalent in λ. Now consider the context C^{ref} that implements a counter using a reference cell. The fully abstract compiler would have to disallow linking between e_1 and e_2 with C^{ref} since this linking disrupts the equivalence. What if the programmer wants to link these together and is willing to give up some equivalences in order to do so?

We present linking-types extension for λ^{κ} and $\lambda^{\text{ref}^{\kappa}}$ to enable such linking. The λ^{κ} type system includes reference types and tracks heap effects. It also includes a computation type $R^{\epsilon}\tau$ that will be compiled to the target computation type $E^{\epsilon}_{\text{exn}}\tau$. Additionally, we provide type conversion functions κ^{+} and κ^{-} to relate types in λ and λ^{κ}.

With this extension, the programmer can annotate e_1 and e_2 with a linking type that specifies that the input to these programs can be heap-effecting. At this type, e_1 and e_2 are no longer contextually equivalent and further can be linked with λ^{ref}'s the counter library.

Without the annotation, the compiler would translate the types of e_1 and e_2 which is the λ type `unit -> int` to the $\lambda^{\text{ref}}_{\text{exn}}$ type `unit ->`E^0_O and the type of counter which is the λ^{ref} type `unit -> int` to the $\lambda^{\text{ref}}_{\text{exn}}$ type `unit ->` E^1_O type. Since these types are not the same an error would be reported and linking can't happen.

By contrast, λ^{κ}'s type (`unit -> R^1 int) -> int` that the programmer annotate e_1 and e_2 with will be translated to `(unit -> E^1_0 int) -> E^1_0 int` which is the same from the translation of the counter library.