Date and Time: 10/24/2017, Tuesday at 5:00 - 6:00
Presenter: Hyeyoung Shin

Intro

This talk is based on the paper *Linking Types for Multi-language Software* by Daniel Patterson and Amal Ahmed, which is presented at SNAPL 2017.

Motivation

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 are inexpressible in their source language but are 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.

Previous Works

There have been efforts to solve this problem. A 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 are also a possible solution, but with a limitation that programmers need to understand the full intermediate ST language and the compiler from R to T.

Our Solution: Linking Types

The proposed solution in this paper is to extend source language specifications with linking types. This is a better solution than previous works because they minimally enrich source language types and give programmers fine-grained control by letting them annotate individual terms that need linking.

Linking Types Formally

To introduce linking types formally, we consider two simple source languages, λ and λ^{ref}. λ is the simply typed lambda calculus with unit and 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 $\lambda^{\text{ref}}_{\text{exn}}$. 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 can be used to rule out linking λ with the λ^{ref} components that use mutable references. We also add exceptions to the target to represent the extra control flow commonly found in low-level languages.

Examples

Let me illustrate the linking idea with two example programs e_1 and e_2. These two programs are equivalent in λ. Since λ is pure, calling c once or twice produces the same result. Now consider the context C^{ref} that implements a counter using a reference cell. The fully abstract compiler would have to disallow linking with C^{ref^+} because it can distinguish them. In other words, e_1^+ and e_2^+ are no longer contextually equivalent. What if the programmer wants to link these together and is willing to give up some equivalences in order to do so?

To enable this linking, we present linking-types extensions, λ^κ and $\lambda^{\text{ref}^\kappa}$. The λ^κ type system includes the reference type and the computation type, $R^\epsilon\,\tau$, which is analogous to $E^\epsilon\,\tau$ in the target type system. We will use $R^\epsilon\,\tau$ to track heap effects.

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 $\lambda^{\text{ref}^\kappa}$'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 c' which is the λ^{ref} type **unit -> int** to the $\lambda^{\text{ref}}_{\text{exn}}$ type **unit ->** $E_0^{\dot{}}$ 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 ->** $R^{\dot{}}$ **int -> int** that the programmer annotate e_1 and e_2 with will be translated to **(unit ->** $E_{\dot{O}}$ **int) ->** $E_{\dot{O}}$ **int** which is the same from the translation of the counter library.

Please note that we are not changing the programming language itself. The terms added in the extension are there only for programmer's reasoning. (so that the programmer is aware of the contexts in which his/her programs are linked) Linking types should only allow a programmer to change equivalence of their existing language.

Additionally, to relate types of λ and λ^κ the linking types extension is equipped with type conversion functions κ^+ and κ^-. We will discuss this in more detail in the properties of linking types.

Properties of Linking Types

For any source language λ_{src}, an extended language λ^{κ}_{src} with type converting functions κ^+ and κ^- is a linking types extension if the following properties hold:

- $\forall e \in \lambda_{src}. e \in \lambda^{\kappa}_{src}$
- $\forall \tau \in \lambda_{src}. \kappa^+(\tau) = \tau^{\kappa}$
- $\forall \tau^{\kappa} \in \lambda^{\kappa}_{src}. \kappa^-(\tau^{\kappa}) = \tau$
- $\forall \tau \in \lambda_{src}. \kappa^-(\kappa^+(\tau)) = \tau$
- $\forall e_1, e_2 \in \lambda_{src}. e_1 \approx^{ctx}_{\lambda_{src}} e_2 : \tau \iff e_1 \approx^{ctx}_{\lambda^{\kappa}_{src}} e_2 : \kappa^+(\tau)$
- The set of programs you can write in λ_{src} is the same set of programs you can write in λ^{κ}_{src}.

1. λ_{src} terms are a subset of λ^{κ}_{src} terms.
2. λ_{src} type τ embeds into a λ^{κ}_{src} type by $\kappa^+(\tau)$.
3. λ^{κ}_{src} type τ^{κ} projects to a λ_{src} type by $\kappa^-(\tau^{\kappa})$.
4. For any λ_{src} type $\tau, \kappa^-(\kappa^+(\tau)) = \tau$.
5. κ^+ preserves and reflects equivalence.
6. The same.

We can understand the effect and properties of linking types by studying how the extension affect equivalence classes of programs using example program A, B, and C.

$A : \lambda f : int \to int. 1$
$B : \lambda f : int \to int. f0; 1$
$C : \lambda f : int \to int. f0; f0; 1$

At the type **(int -> int) -> int**, these three programs are equivalent in λ, which is illustrated by A,B,C in a single equivalence box. In λ^{ref}, f may increment a counter, thus A,B,C could have externally visible effects. So they are in separate boxes.

Now we consider different type annotations a programmer of $\lambda^{ref^{\kappa}}$ can give to A, B, and C. Using the computation type $R^{\epsilon}\, int$, we can convert **(int -> int) -> int** into four different types:

(int -> $R^{o}\, int$) -> $R^{o}\, int$
(int -> $R^{o}\, int$) -> $R^{\cdot}\, int$
(int -> $R^{\cdot}\, int$) -> $R^{o}\, int$
(int -> $R^{\cdot}\, int$) -> $R^{\cdot}\, int$

Observe the changed made in the equivalence classes in $\lambda^{ref^{\kappa}}$. At the type (int -> $R^{o}\, int$) -> $R^{o}\, int$, A, B, and C are all equivalent since this linking type requires that f be pure. At the type (int -> $R^{\cdot}\, int$) -> $R^{\cdot}\, int$ all three programs are in different equivalence classes, because the linking type allows f to be impure. At the type (int -> $R^{o}\, int$) -> $R^{\cdot}\, int$ all three programs are equivalent again. Since f is pure, allowing the body to be impure does not make any difference. The last linking type (int -> $R^{\cdot}\, int$) -> $R^{o}\, int$ can only be assigned to A, because an impure f could not have been called to produce a pure result.

The λ programmer who wants to link one of these programs with a λ^{ref} program can use λ^{κ}'s type (int -> $R^{\cdot}$ *int*) -> $R^{\cdot}$ *int* to annotate the function f to enable linking. In this process, he chooses to lose the equivalence in order to link.