“Verifying an Open Compiler Using Multi-Language Semantics”

By James T. Perconti and Amal Ahmed

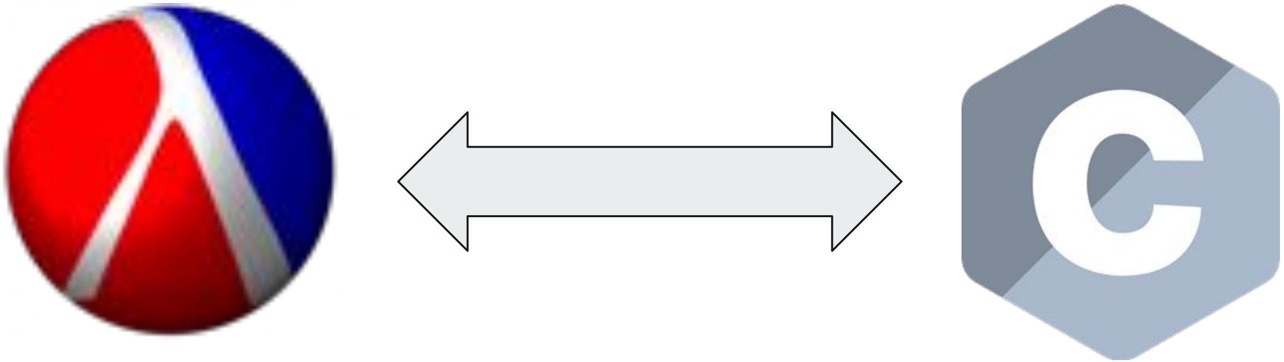
Presentation By Kevin Cam, Noble Mushtak, and Anthony Mu

Problem

- Projects are developed in multiple languages.
- However, the way interoperability between different programming languages is implemented is very “unsafe.”

Example

- Let’s say you are developing a game in both Racket and C.
- How do you ensure that the C code does not violate the structure of data from Racket?



Goal

- Therefore, we would like to be able to **prove** that when we link two programs from different programming languages together, they work together in the manner that was intended.

Related Work: Benton-Hur Approach

- Benton-Hur 2009
- Relies on creating a logical relation between the source code and the target code
    - Two programs are *logically related* if they have the same semantics or behavior.
- Drawbacks: Vertical and Horizontal Compositionality
    - Vertical: Doesn’t scale to a multi-pass compiler
    - Horizontal: Approach is very limited to simple components

The Paper

- “Verifying an Open Compiler Using Multi-Language Semantics”
- James T. Perconti and Amal Ahmed
- 23rd European Symposium on Programming (ESOP 2014)
- Proposed methodology on proving *compositional* compiler correctness

Compiler Correctness

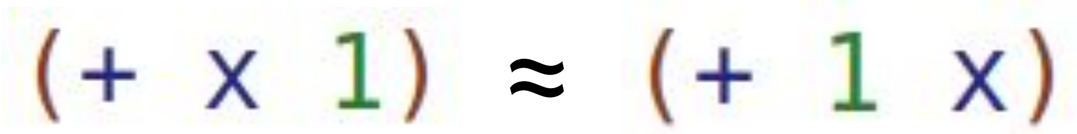
- A *compiler* translates a program in one programming language into another programming language.
- A compiler is *correct* if the program it outputs always has the same *behavior* as the original program.
- First successful implementation was the CompCert C compiler
    - LeRoy 2006
    - Drawback: Only worked on *closed* programs

Compositional Compiler Correctness

- An *open* compiler is a compiler that translates *open* programs into another programming language.
- *Compositional* compiler correctness is the problem of proving that open compilers are correct.
- However, how do we prove that two open programs have the same behavior, if we can’t run open programs?

Contextual Equivalence

- Two open programs e_1 and e_2 are *contextually equivalent* if, for any closed program which contains e_1, we can replace e_1 with e_2 and the behavior of the closed program will not change.
- Unlike logical relations, contextual equivalence is only defined for programs *written in the same programming language*.



```
(+ x 1) ≈ (+ 1 x)
```

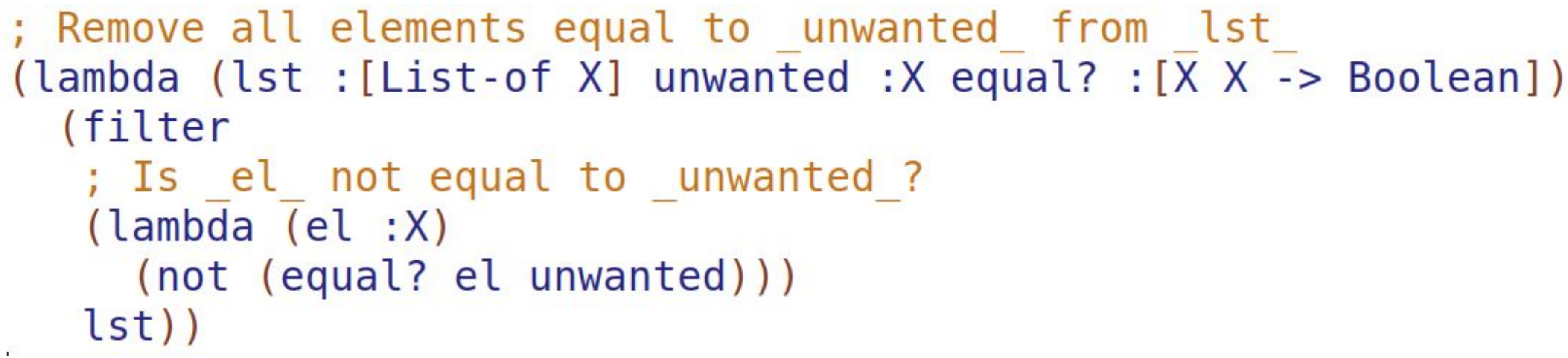
Multi-Language Operational Model

- Three languages:
    - F (System F): Functional, Has Closures
    - C (Closure Conversion): Functional, but No Closures
    - A (Allocation): Allows Mutation
- FCA is a language which incorporates all three of the above programming languages.

System F

- Independently created by Jean-Yves Girard, in 1972, and John C. Reynolds, in 1974
- Very similar to functional programming languages like ML
- Statically typed
- Allows for higher-order functions, closures
- Universal types (i.e. generic data definitions)
- Recursive types (i.e. recursive data definitions)

Example of F Code

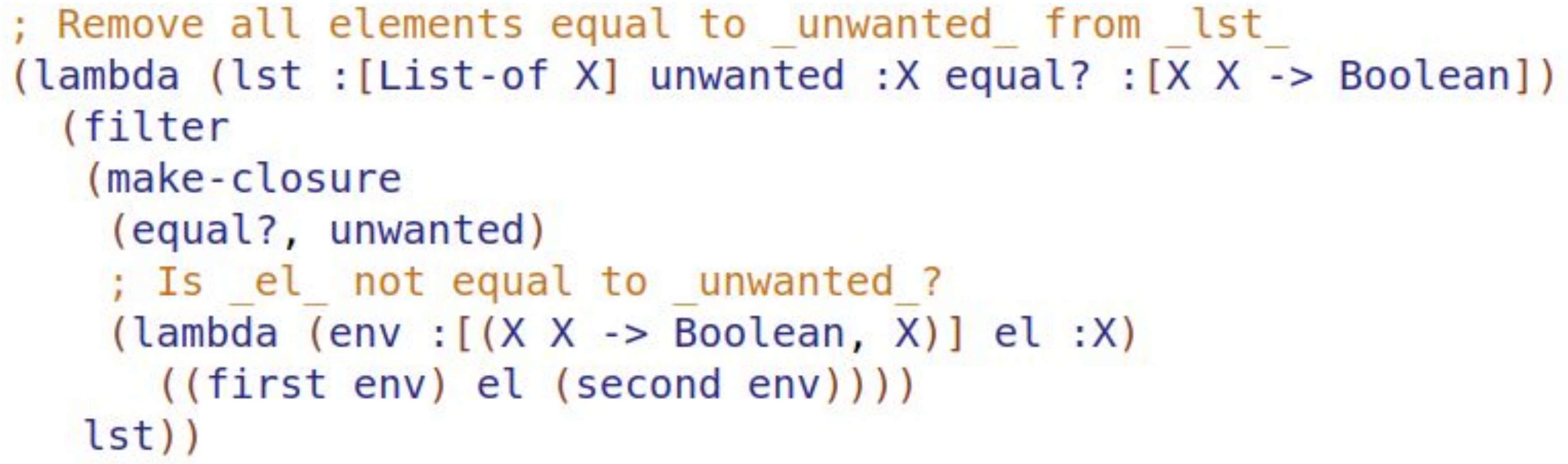


```
; Remove all elements equal to _unwanted_ from _lst_
(lambda (lst :[List-of X] unwanted :X equal? :[X X -> Boolean])
  (filter
   ; Is _el_ not equal to _unwanted_?
   (lambda (el :X)
     (not (equal? el unwanted)))
   lst))
```

The Compiler (F to C)

- The main difference between F and C is that C does not allow for free variables inside of functions.
  - Therefore, the free variables associated with each function in F code are encapsulated in a tuple called an *environment*, which is then passed as an argument to the function.
  - This closed function is paired with the environment tuple to create a closure.

Example of C code

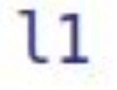


```
; Remove all elements equal to _unwanted_ from _lst_
(lambda (lst :[List-of X] unwanted :X equal? :[X X -> Boolean])
  (filter
   (make-closure
    (equal?, unwanted)
    ; Is _el_ not equal to _unwanted_?
    (lambda (env :[(X X -> Boolean, X)] el :X)
      ((first env) el (second env))))
   lst))
```

The Compiler (C to A)

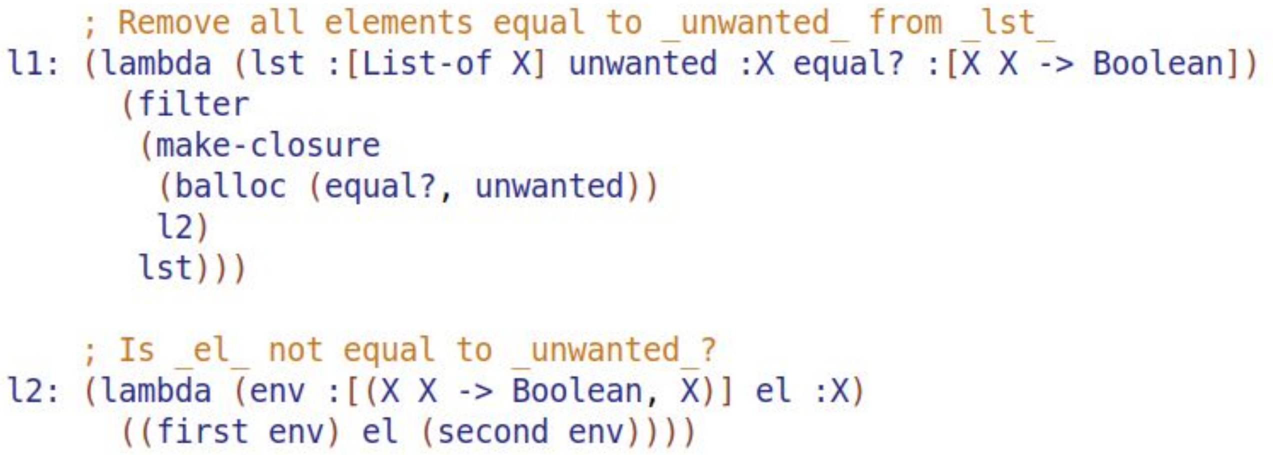
- The main difference between C and A is that A has *memory allocation*.
- In order to accomodate with A code's memory allocation, we create a *memory heap* for the translated C component.
  - All functions and tuples are stored on the memory heap.
  - The compiler translates C tuples by generating *balloc* expressions, which represents dynamic memory allocation.

Example of A Code

Program: 

```
l1
```

Memory Heap:



```
    ; Remove all elements equal to _unwanted_ from _lst_
l1: (lambda (lst :[List-of X] unwanted :X equal? :[X X -> Boolean])
      (filter
       (make-closure
        (balloc (equal?, unwanted))
        l2)
       lst)))

    ; Is _el_ not equal to _unwanted_?
l2: (lambda (env :[(X X -> Boolean, X)] el :X)
      ((first env) el (second env))))
```

FCA: Interoperability Language

- **Boundary Terms** are used to insert a fragment of code from one language into code from another language
- FC(e_C) allows e_C, a program in C code, to be treated like a program in F code
- CF(e_F) allows e_F, a program in F code, to be treated like a program in C code
- There are similar boundary terms for CA and AC

$\mathcal{CF}^{\tau}\,e$

Lump Types

$L\langle\tau\rangle$

- **Lump types** are used to convert C types into opaque F types, and convert A types to opaque C types.
  - When translating closure values in C to F, we use lump types in order to encode the type of the environment, so that the environment of the closure remains opaque to F.
  - Introduced by Matthews and Findler in 2007

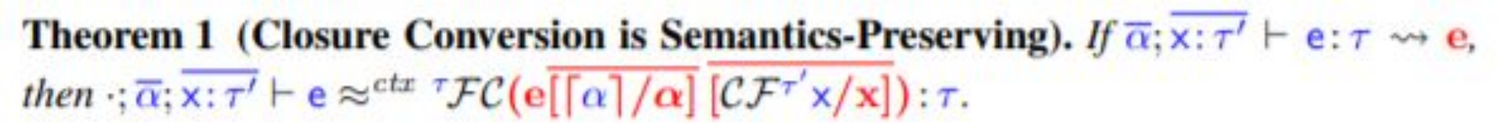
Suspension Types

$\lceil \alpha \rceil$

- **Suspensions** are used to convert F type variables into a C type variable, and C to A
  - Only used on type variables, unlike lump types
  - Used when C needs access to type variables from F, or when A needs access to type variables from C
  - Suspensions are a unique concept introduced by Perconti and Ahmed in this paper

Compiler Correctness Theorem: F to C

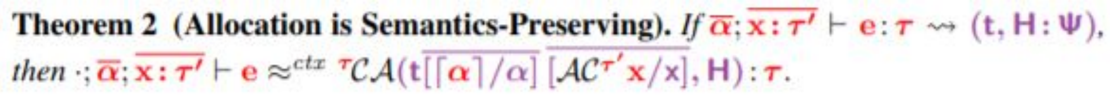
- Between F and C:
    - If the open program e_F in F compiles to the program e_C in C, then the original program e_F in FCA is contextually equivalent to FC(e_C)



Theorem 1 (Closure Conversion is Semantics-Preserving). *If* $\overline{\alpha}; \overline{\mathsf{x} : \tau'} \vdash \mathsf{e} : \tau \rightsquigarrow \mathbf{e}$, *then* $\cdot; \overline{\alpha}; \overline{\mathsf{x} : \tau'} \vdash \mathsf{e} \approx^{ctx} {}^{\tau}\mathcal{FC}(\mathbf{e}\overline{[\lceil\alpha\rceil/\boldsymbol{\alpha}]}\,\overline{[\mathcal{CF}^{\tau'}\mathsf{x}/\mathbf{x}]}) : \tau$.

Compiler Correctness Theorem: C to A

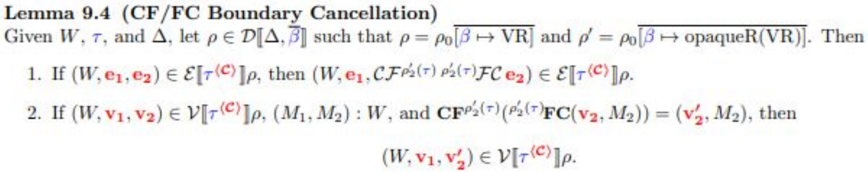
- Between C and A:
  - If the open program e_C in C compiles to the program (e_A, H) in A, then the original program e_C in FCA is contextually equivalent to $CA((e_A, H))$
- Since contextual equivalence is transitive, this also proves compiler correctness between F and A.



Theorem 2 **(Allocation is Semantics-Preserving).** *If* $\overline{\alpha}; \overline{\mathtt{x} : \tau'} \vdash \mathtt{e} : \tau \rightsquigarrow (\mathtt{t}, \mathtt{H} : \Psi)$, *then* $\cdot; \overline{\alpha}; \overline{\mathtt{x} : \tau'} \vdash \mathtt{e} \approx^{ctx} {}^{\tau}\mathcal{CA}(\mathtt{t}\overline{[\lceil \alpha \rceil / \alpha]}\ \overline{[\mathcal{AC}^{\tau'}\mathtt{x}/\mathtt{x}]}, \mathtt{H}) : \tau$.

Proving Compiler Correctness

- To prove the compiler correctness theorem, they design a “step-indexed Kripke logical relation” as a sound and complete model of contextual equivalence in FCA.
  - The Kripke logical relation is defined on both programs and memory heaps.
  - This kind of logical relation makes it easy to handle recursive types and memory allocation.



Lemma 9.4 (CF/FC Boundary Cancellation)
Given W, τ, and Δ, let $\rho \in \mathcal{D}[\![\Delta, \overline{\beta}]\!]$ such that $\rho = \rho_0\overline{[\beta \mapsto \mathrm{VR}]}$ and $\rho' = \rho_0\overline{[\beta \mapsto \mathrm{opaqueR(VR)}]}$. Then

1. If $(W, \mathbf{e_1}, \mathbf{e_2}) \in \mathcal{E}[\![\tau^{\langle \mathbf{C} \rangle}]\!]\rho$, then $(W, \mathbf{e_1}, \mathcal{CF}^{\rho_2'(\tau)}\, {}^{\rho_2'(\tau)}\mathcal{FC}\,\mathbf{e_2}) \in \mathcal{E}[\![\tau^{\langle \mathbf{C} \rangle}]\!]\rho$.
2. If $(W, \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau^{\langle \mathbf{C} \rangle}]\!]\rho$, $(M_1, M_2) : W$, and $\mathbf{CF}^{\rho_2'(\tau)}({}^{\rho_2'(\tau)}\mathbf{FC}(\mathbf{v_2}, M_2)) = (\mathbf{v_2'}, M_2)$, then
$$(W, \mathbf{v_1}, \mathbf{v_2'}) \in \mathcal{V}[\![\tau^{\langle \mathbf{C} \rangle}]\!]\rho.$$

Proving Compiler Correctness (cont.)

- The two properties vital for this proof are the bridge lemma and boundary cancellation.
  - Bridge Lemma: For any two values v_1 and v_2 in F, if v_1 and v_2 are logically related, then CF(v_1) and CF(v_2) are logically related.
  - Boundary Cancellation: Any program e_C written in C code is contextually equivalent to CF(FC(e_C)) in FCA.

Discussion and Future Work

- Getting Rid of Suspension Types
- Compiling To Assembly
- Mutable References
- Adding Compiler Passes
- Supporting Realistic Interoperability

References

- Benton, N., Hur, C.K.: Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, Apr. 2010.
- Cardelli, L.: Type Systems. Microsoft Research, Feb. 2004.
- Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In POPL 2007.
- Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In POPL 2006.
- J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In European Symposium on Programming (ESOP), Apr. 2014