Programming Up-to-Congruence, Again

Stephanie Weirich

University of Pennsylvania

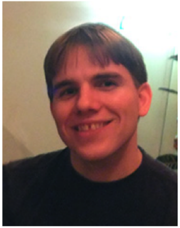
August 12, 2014

WG 2.8 Estes Park

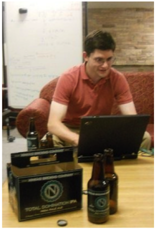
Zombie

A functional programming language with a dependent type system intended for "lightweight" verification

With:



Vilhelm Sjöberg



Chris Casinghino

plus Trellys team (Aaron Stump, Tim Sheard, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell)

Zombie language

- Support for both functional programming (including nontermination) and reasoning in constructive logic
- Full-spectrum dependent-types (for uniformity)
- Erasable arguments (for efficient compilation)
- Simple semantics for dependently-typed pattern matching
- Proof automation based on congruence closure

Nongoal: mathematical foundations, full program verification

Zombie: A language, in two parts

1. Logical fragment: all programs must terminate (similar to Coq and Agda)

```
log add : Nat → Nat → Nat
ind add x y = case x [eq] of
   Zero   → y                 -- eq : x = Zero
   Suc x’ → add x’ [ord eq] y -- eq : x = Suc x’, used for ind
```

Zombie: A language, in two parts

1. Logical fragment: all programs must terminate (similar to Coq and Agda)

```
log add : Nat → Nat → Nat
ind add x y = case x [eq] of
   Zero   → y                  -- eq : x = Zero
   Suc x’ → add x’ [ord eq] y -- eq : x = Suc x’, used for ind
```

2. Programmatic fragment: nontermination allowed

```
prog div : Nat → Nat → Nat
rec div n m = if n < m then 0 else 1 + div (n - m) m
```

Zombie: A language, in two parts

1. Logical fragment: all programs must terminate (similar to Coq and Agda)

```
log add : Nat → Nat → Nat
ind add x y = case x [eq] of
   Zero   → y                   -- eq : x = Zero
   Suc x' → add x' [ord eq] y -- eq : x = Suc x', used for ind
```

2. Programmatic fragment: nontermination allowed

```
prog div : Nat → Nat → Nat
rec div n m = if n < m then 0 else 1 + div (n - m) m
```

Uniformity: Both fragments use the same syntax, have the same (call-by-value) operational semantics.

One type system for two fragments

Typing judgement specifies the fragment (where $\theta = \mathsf{L} \mid \mathsf{P}$)

$$\Gamma \vdash^{\theta} a : A$$

which in turn specifies the properties of the fragment.

Theorem (Type Soundness)

If $\cdot \vdash^{\theta} a : A$ *and if* $a \leadsto^{*} a'$ *then* $\cdot \vdash a' : A$ *and* a' *is a value.*

Theorem (Logical Consistency)

If $\cdot \vdash^{\mathsf{L}} a : A$ *then* $a \leadsto^{*} v$

Reasoning about programs

The logical fragment demands termination, but can reason about the programmatic fragment.

```
log div62 : div 6 2 = 3
log div62 = join
```

(Here `join` is the proof that two terms reduce to the same value.)

Reasoning about programs

The logical fragment demands termination, but can reason about the programmatic fragment.

```
log div62 : div 6 2 = 3
log div62 = join
```

(Here `join` is the proof that two terms reduce to the same value.)

Type checking `join` is undecidable, so includes an overridable timeout.

Type checking without β

The type checker reduces terms *only* when directed by the programmer (e.g. while type checking `join`).

Type checking without β

The type checker reduces terms *only* when directed by the programmer (e.g. while type checking `join`).

ZOMBIE does not include β-convertibility in *definitional equality*!

In a context with

```
f : Vec Nat 3 → Nat
x : Vec Nat (div 6 2)
```

the expression `f x` does **not** type check because `div 6 2` is **not** equal to `3`.

Type checking without β

The type checker reduces terms *only* when directed by the programmer (e.g. while type checking `join`).

Zombie does not include β-convertibility in *definitional equality*!

In a context with

```
f : Vec Nat 3 → Nat
x : Vec Nat (div 6 2)
```

the expression `f x` does **not** type check because `div 6 2` is **not** equal to `3`.

In other words, β-convertibility is only available for *propositional* equality.

Isn't type checking without β awful?

Isn't type checking without β awful?

Yes.

Isn't type checking without β awful?

Yes. And our simple semantics for dependently-typed pattern matching makes it worse.

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
  case n [eq] of
   Zero → (join : 0 + 0 = 0)
              ▷ [~eq + 0 = ~eq]   -- explicit type coercion
                                   -- eq : 0 = n
   Suc m →
     let ih = npluszero m [ord eq] in
       (join : (Suc m) + 0 = Suc (m + 0))
        ▷ [(Suc m) + 0 = Suc ~ih]   -- ih : m + 0 = m
        ▷ [~eq + 0 = ~eq]           -- eq : Suc m = n
```

Isn't type checking without β awful?

Yes. And our simple semantics for dependently-typed pattern matching makes it worse.

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
  case n [eq] of
   Zero → (join : 0 + 0 = 0)
              ▷ [~eq + 0 = ~eq]   -- explicit type coercion
                                   -- eq : 0 = n
   Suc m →
     let ih = npluszero m [ord eq] in
       (join : (Suc m) + 0 = Suc (m + 0))
         ▷ [(Suc m) + 0 = Suc ~ih]   -- ih : m + 0 = m
         ▷ [~eq + 0 = ~eq]           -- eq : Suc m = n
```

But we can do better.

Opportunity: Congruence Closure

What if we base definitional equivalence on the *congruence closure* of equations in the context?

$$\frac{x : a = b \in \Gamma}{\Gamma \vdash a = b} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash \{a/x\}\, c = \{b/x\}\, c}$$

$$\frac{}{\Gamma \vdash a = a} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash b = a} \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash b = c}{\Gamma \vdash a = c}$$

Efficient algorithms for deciding this relation exist [Nieuwenhuis and Oliveras, 2007].
But, extending this relation with β-conversion makes it undecidable.

Example with CC

The type checker automatically takes advantage of equations in the context.

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
   case n [eq] of
      Zero → (join : 0 + 0 = 0)
         -- coercion by eq inferred
      Suc m →
        let ih = npluszero m [ord eq] in
          (join : (Suc m) + 0 = Suc (m + 0))
          -- coercion by eq and ih inferred
```

ZOMBIE language design

- Semantics defined by an explicitly-typed **core language** [Casinghino et al. POPL '14][Sjöberg et al., MSFP'12]
  - Definitional equality is α-equivalence (no CC)
  - All uses of propositional equality must be explicit
  - Core language is type sound

Zombie language design

- Semantics defined by an explicitly-typed **core language** [Casinghino et al. POPL '14][Sjöberg et al., MSFP'12]
  - Definitional equality is α-equivalence (no CC)
  - All uses of propositional equality must be explicit
  - Core language is type sound
- Concise **surface language** for programmers [Sjöberg and Weirich, draft paper]
  - Specified via bidirectional type system
  - Definitional equality is Congruence Closure
  - Elaborates to core language

Zombie language design

- Semantics defined by an explicitly-typed **core language** [Casinghino et al. POPL '14][Sjöberg et al., MSFP'12]
  - Definitional equality is α-equivalence (no CC)
  - All uses of propositional equality must be explicit
  - Core language is type sound
- Concise **surface language** for programmers [Sjöberg and Weirich, draft paper]
  - Specified via bidirectional type system
  - Definitional equality is Congruence Closure
  - Elaborates to core language
- Implementation available, with extensions `https://code.google.com/p/trellys/`

Properties of elaboration

- **Elaboration is sound**
  If elaboration succeeds, it produces a well-typed core language term.
- **Elaboration is complete**
  If a term type checks according to the surface language specification, then elaboration will succeed.
- **Elaboration doesn't change the semantics**
  If elaboration succeeds, it produces a core language term that differs from the source term only in erasable information (type annotations, type coercions, erasable arguments).

Zombie-style Congruence Closure

1. Works up-to-erasure

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}$$

Zombie-style Congruence Closure

1. Works up-to-erasure

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}$$

2. Supports injectivity of type (and data) constructors

$$\frac{\Gamma \vDash ((x : A_1) \to B_1) = ((x : A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}$$

ZOMBIE-style Congruence Closure

1. Works up-to-erasure

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}$$

2. Supports injectivity of type (and data) constructors

$$\frac{\Gamma \vDash ((x : A_1) \to B_1) = ((x : A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}$$

3. Makes use of assumptions that are *equivalent* to equalities

$$\frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}$$

ZOMBIE-style Congruence Closure

1. Works up-to-erasure

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}$$

2. Supports injectivity of type (and data) constructors

$$\frac{\Gamma \vDash ((x : A_1) \to B_1) = ((x : A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}$$

3. Makes use of assumptions that are *equivalent* to equalities

$$\frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}$$

4. Only includes typed terms

Zombie-style Congruence Closure

1. Works up-to-erasure

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}$$

2. Supports injectivity of type (and data) constructors

$$\frac{\Gamma \vDash ((x : A_1) \to B_1) = ((x : A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}$$

3. Makes use of assumptions that are *equivalent* to equalities

$$\frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}$$

4. Only includes typed terms
5. and generates proof terms in the core language

Examples and Extensions

Proof inference

Congruence closure can supply proofs of equality

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
  case n [eq] of
    Zero →
      let _ = (join : 0 + 0 = 0) in _
    Suc m →
      let _ = npluszero m [ord eq] in
      let _ = (join : (Suc m) + 0 = Suc (m + 0)) in _
```

Extension: Unfold

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
  case n [eq] of
    Zero  → unfold (0 + 0) in _
    Suc m →
      let _ = npluszero m [ord eq] in
      unfold ((Suc m) + 0) in _
```

The expression `unfold a in b` expands to

```
let _ = (join : a = a1)   in
let _ = (join : a1 = ...) in
...
let _ = (join : ... = an) in
  b
```

when `a` ⇝ `a1` ⇝ `...` ⇝ `an`

Extension: Reduction Modulo

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
   case n [eq] of
      Zero → unfold (n + 0) in _
      Suc m →
        let ih = npluszero m [ord eq] in
        unfold (n + 0) in _
```

The type checker makes use of congruence closure when reducing terms with `unfold`.
E.g., if we have $h : n = \mathsf{0}$ in the context, allow the step

$$n + 0 \leadsto_{\mathsf{cbv}} 0$$

Extension: Smart join

```
log npluszero : (n : Nat) → (n + 0 = n)
ind npluszero n =
   case n [eq] of
      Zero → smartjoin
      Suc m →
        let ih = npluszero m [ord eq] in
        smartjoin
```

Use unfold (and reduction modulo) on both sides of an equality when type checking `join`.

Smart case

An Agda Puzzle

Consider an operation that appends elements to the end of a list.

```
snoc : List → A → List
snoc xs x = xs ++ (x :: [])
```

How would you prove the following property in Agda?

```
snoc-inv : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv (x :: xs')   (y :: ys') z pf = ?
...
```

An Agda Puzzle

Consider and operation that appends elements to the end of a list.

```
snoc : List → A → List
snoc xs x = xs ++ x :: []
```

How would you prove the following property in Agda?

```
snoc-inv : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv (x :: xs')   (y :: ys') z pf with (snoc xs' z) | (snoc ys' z)
         | inspect (snoc xs') z | inspect (snoc ys') z
snoc-inv (.y :: xs')  (y :: ys') z refl | .s | s
         | [ p ] | [ q ] with (snoc-inv xs' ys' z (trans p (sym q)))
snoc-inv (.y :: .ys') (y :: ys') z refl | .s | s
         | [ p ] | [ q ] | refl = refl
...
```

Uses Agda idiom called “inspect on steroids.”

Smart case

Zombie solution is more straightforward:

```
log snoc_inv : (xs ys: List A) → (z : A)
            → (snoc xs z) = (snoc ys z) → xs = ys
ind snoc_inv xs ys z pf =
     case xs [eq], ys of
        Cons x xs' , Cons y ys' →
           let _ = smartjoin : snoc xs z = Cons x (snoc xs' z) in
           let _ = smartjoin : snoc ys z = Cons y (snoc ys' z) in
           let _ = snoc_inv xs' [ord eq] ys' z _ in
           _
        ...
```

Pattern matching introduces equalities (like `eq`) into the context in each branch. CC takes advantage of them automatically.

Conclusion and Future Work

- We should be thinking about the combination of dependently-typed languages and nontermination.

Conclusion and Future Work

- We should be thinking about the combination of dependently-typed languages and nontermination.
- Restriction on β-reduction leads us to the exploration of alternative forms of definitional equality, specifically congruence closure

Conclusion and Future Work

- We should be thinking about the combination of dependently-typed languages and nontermination.
- Restriction on β-reduction leads us to the exploration of alternative forms of definitional equality, specifically congruence closure
- Congruence closure powers smart case, a simple specification of dependently-typed pattern matching

Conclusion and Future Work

- We should be thinking about the combination of dependently-typed languages and nontermination.
- Restriction on β-reduction leads us to the exploration of alternative forms of definitional equality, specifically congruence closure
- Congruence closure powers smart case, a simple specification of dependently-typed pattern matching
- Proof automation is an important part of the design of dependently-typed languages, but should be backed up by specifications