Programming Up-to-Congruence

Vilhelm Sjöberg and Stephanie Weirich

University of Pennsylvania

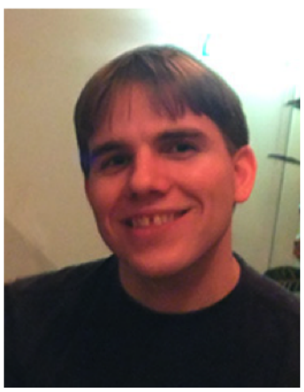
January 16, 2015

POPL 2015

Zombie

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

With:



Vilhelm Sjöberg
Yale University

Chris Casinghino
Draper Labs

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

The Zombie programming language

Goal: FP++

- Functional programming enhanced by reasoning in constructive logic
- Full-spectrum dependent types (for uniformity)
- Erasable arguments (for efficient compilation)
- Simple semantics for indexed types and dependently-typed pattern matching

The Zombie programming language

Goal: FP++

- Functional programming enhanced by reasoning in constructive logic
- Full-spectrum dependent types (for uniformity)
- Erasable arguments (for efficient compilation)
- Simple semantics for indexed types and dependently-typed pattern matching
- **Proof automation based on congruence closure**

Zombie: A language, in two parts

1. Programmatic fragment: nontermination allowed (similar to ML and Haskell)

```
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. Programmatic fragment: nontermination allowed (similar to ML and Haskell)

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

2. 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. Programmatic fragment: nontermination allowed (similar to ML and Haskell)

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

2. 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
```

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

Dependent types in Zombie

The logical fragment can reason about the programmatic fragment.

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

Here, `join` proves that two terms are equal because they reduce to the same value.

Dependent types in Zombie

The logical fragment can reason about the programmatic fragment.

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

Here, `join` proves that two terms are equal because they reduce to the same value.

Type checking `join` is undecidable, so includes an overridable timeout—the programmer is in control.

Restricted β-equality

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

Restricted β-equality

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

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

Restricted β-equality

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

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

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

```
f (x |> [Vec Bool ~div62])
```

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.

Better

What if the type checker could determine those coercions automatically?

```
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
```

i.e. automatically coerce type `0 + 0 = 0` to type `n + 0 = n` in contexts where `eq: n = 0` is assumed.

Better

What if the type checker could determine those coercions automatically?

```
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
```

i.e. automatically coerce type `0 + 0 = 0` to type `n + 0 = n` in contexts where `eq: n = 0` is assumed.

Capture this idea with a relation:

$$\texttt{eq: n = 0} \vdash \big(\texttt{0 + 0 = 0}\big) = \big(\texttt{n + 0 = n}\big)$$

Opportunity: Congruence Closure

The relation that we need is 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].
Note, extending this relation with β-conversion makes it undecidable.

What we have done

Designed and implemented a concise **surface language** for ZOMBIE programmers

- Specification via bidirectional type system

$$\Gamma \vdash a \Rightarrow A \quad \text{and} \quad \Gamma \vdash a \Leftarrow A$$

- Type checking is up-to Congruence Closure

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vDash A = B}{\Gamma \vdash a \Rightarrow B} \qquad \frac{\Gamma \vdash a \Leftarrow A \quad \Gamma \vDash A = B}{\Gamma \vdash a \Leftarrow B}$$

- Elaborates to explicitly-typed **core language**, previously proven sound
  [POPL '14][MSFP'12]

ZOMBIE-style Congruence Closure

1. Only includes well-typed terms

Zombie-style Congruence Closure

1. Only includes well-typed terms
2. 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. Only includes well-typed terms
2. Makes use of assumptions that are *equivalent* to equalities

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

3. 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. Only includes well-typed terms
2. Makes use of assumptions that are *equivalent* to equalities

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

3. Supports injectivity of type (and data) constructors

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

4. 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. Only includes well-typed terms
2. Makes use of assumptions that are *equivalent* to equalities

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

3. Supports injectivity of type (and data) constructors

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

4. Works up-to-erasure

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

5. and generates proof terms in the core language

Properties of Elaboration

- **Elaboration is sound**
  If elaboration succeeds, it produces a well-typed core language term.

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.

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 irrelevant information (type annotations, type coercions, erasable arguments).

Extensions

Proof inference

Congruence closure can also 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

Common to reduce terms as much as possible

```
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

The type checker makes use of congruence closure when reducing terms with `unfold`.

```
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 _
```

E.g., if we have $h : n = \mathsf{0}$ in the context, allow the step

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

Extension: Smartjoin

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

Conclusions

- Dependently-typed languages should allow nonterminating programs, but compile-time reduction is tricky
- Restricting β-reduction allows alternative forms of automatic reasoning, 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, and should be backed up by specifications

Implementation and examples available:

`https://code.google.com/p/trellys/source/browse/trunk/zombie-trellys/`

or Google: zombie trellys

Thanks!