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)

\[
\begin{align*}
\text{log} \quad \text{add} & : \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{ind} \quad \text{add} \ x \ y & = \text{case} \ x \ [\text{eq}] \ of \\
& \quad \text{Zero} \quad \to \quad y \quad -- \ \text{eq} : x = \text{Zero} \\
& \quad \text{Suc} \ x' \quad \to \quad \text{add} \ x' \ [\text{ord} \ \text{eq}] \ y \quad -- \ \text{eq} : x = \text{Suc} \ x', \ used \ for \ \text{ind}
\end{align*}
\]
ZOMBIE: A language, in two parts

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

   \[
   \text{log add} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}
   \]
   \[
   \text{ind add } x \ y = \text{case } x \ [\text{eq}] \ of
   \]
   \[
   \text{Zero } \rightarrow y \quad \-- \text{eq} : x = \text{Zero}
   \]
   \[
   \text{Suc } x' \rightarrow \text{add } x' \ [\text{ord eq}] \ y \quad \-- \text{eq} : x = \text{Suc } x', \text{ used for ind}
   \]

2. Programmatic fragment: nontermination allowed

   \[
   \text{prog div} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}
   \]
   \[
   \text{rec div } n \ m = \text{if } n < m \ \text{then } 0 \ \text{else } 1 + \text{div } (n - m) \ m
   \]
Zombie: A language, in two parts

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

```plaintext
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

```plaintext
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 \( \theta = L \mid P \))

\[ \Gamma \vdash^\theta a : A \]

which in turn specifies the properties of the fragment.

**Theorem (Type Soundness)**

If \( \cdot \vdash^\theta a : A \) and if \( a \rightsquigarrow^* a' \) then \( \cdot \vdash a' : A \) and \( a' \) is a value.

**Theorem (Logical Consistency)**

If \( \cdot \vdash^L a : A \) then \( a \rightsquigarrow^* v \)
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.

\[
\begin{align*}
\text{log } \text{div62} & : \text{div } 6 \ 2 = 3 \\
\text{log } \text{div62} & = \text{join}
\end{align*}
\]

(Here \text{join} is the proof that two terms reduce to the same value.) Type checking \text{join} is undecidable, so includes an overridable timeout.
The type checker reduces terms *only* when directed by the programmer (e.g. while type checking `join`).
Type checking without $\beta$

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

ZOMBIE does not include $\beta$-convertibility in *definitional equality*!

In a context with

\[
\begin{align*}
f : \text{Vec Nat } 3 & \to \text{Nat} \\
x : \text{Vec Nat } (\text{div } 6 2)
\end{align*}
\]

the expression $f \ x$ does **not** type check because $\text{div } 6 2$ is **not** equal to $3$. 
Type checking without $\beta$

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

**ZOMBIE** does not include $\beta$-convertibility in *definitional equality*!

In a context with

\[
\begin{align*}
f &: \text{Vec Nat } 3 \rightarrow \text{Nat} \\
x &: \text{Vec Nat } (\text{div } 6 \ 2)
\end{align*}
\]

the expression $f \ x$ does **not** type check because $\text{div } 6 \ 2$ is **not** equal to 3.

In other words, $\beta$-convertibility is only available for *propositional* equality.
Isn’t type checking without $\beta$ awful?
Isn’t type checking without $\beta$ awful?

Yes.
Isn’t type checking without $\beta$ awful?

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

\begin{verbatim}
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 = ~eq] -- 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
\end{verbatim}
Isn’t type checking without $\beta$ 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 = ~eq] -- 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?

\[
\begin{align*}
x : a = b & \in \Gamma \\
\Gamma \vdash a = b
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a = b \\
\Gamma \vdash \{a/x\} c = \{b/x\} c
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a = b \\
\Gamma \vdash b = a
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a = b \\
\Gamma \vdash b = c
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a = c
\end{align*}
\]

Efficient algorithms for deciding this relation exist [Nieuwenhuis and Oliveras, 2007].
But, extending this relation with $\beta$-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 $\alpha$-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 $\alpha$-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/
ZOMBIE language design

- Semantics defined by an explicitly-typed **core language** [Casinghino et al. POPL ’14][Sjöberg et al., MSFP’12]
  - Definitional equality is $\alpha$-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

\[
|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B \\
\hline
\Gamma \models a = b
\]

Supports injectivity of type (and data) constructors

\[
\Gamma \models (x : A_1 \rightarrow B_1) = (x : A_2 \rightarrow B_2)
\]

Makes use of assumptions that are equivalent to equalities

\[
x : A \in \Gamma \quad \Gamma \models A_1 = (a = b) \quad \Gamma \models a = b
\]

Only includes typed terms and generates proof terms in the core language
ZOMBIE-style Congruence Closure

1. Works up-to-erasure

\[ |a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B \]
\[ \Gamma \vdash a = b \]

2. Supports injectivity of type (and data) constructors

\[ \Gamma \vdash ((x:A_1) \to B_1) = ((x:A_2) \to B_2) \]
\[ \Gamma \vdash A_1 = A_2 \]
ZOMBIE-style Congruence Closure

1. Works up-to-erasure

\[ |a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B \]
\[ \Gamma \vdash a = b \]

2. Supports injectivity of type (and data) constructors

\[ \Gamma \vdash ((x : A_1) \to B_1) = ((x : A_2) \to B_2) \]
\[ \Gamma \vdash A_1 = A_2 \]

3. Makes use of assumptions that are equivalent to equalities

\[ x : A \in \Gamma \quad \Gamma \vdash A = (a = b) \]
\[ \Gamma \vdash a = b \]
ZOMBIE-style Congruence Closure

1. Works up-to-erasure

\[
\begin{align*}
|a| &= |b| & \Gamma \vdash a : A & \quad \Gamma \vdash b : B \\
\hline
\Gamma \vdash a = b
\end{align*}
\]

2. Supports injectivity of type (and data) constructors

\[
\begin{align*}
\Gamma \vdash ((x:A_1) \to B_1) &= ((x:A_2) \to B_2) \\
\hline
\Gamma \vdash A_1 = A_2
\end{align*}
\]

3. Makes use of assumptions that are equivalent to equalities

\[
\begin{align*}
x : A \in \Gamma & \quad \Gamma \vdash A = (a = b) \\
\hline
\Gamma \vdash a = b
\end{align*}
\]

4. Only includes typed terms
ZOMBIE-style Congruence Closure

1. Works up-to-erasure

\[
|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B \\
\Gamma \vdash a = b
\]

2. Supports injectivity of type (and data) constructors

\[
\Gamma \vdash ((x : A_1) \to B_1) = ((x : A_2) \to B_2) \\
\Gamma \vdash A_1 = A_2
\]

3. Makes use of assumptions that are equivalent to equalities

\[
x : A \in \Gamma \quad \Gamma \vdash A = (a = b) \\
\Gamma \vdash a = b
\]

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

\[
\text{log npluszero : } (n : \text{Nat}) \rightarrow (n + 0 = n)
\]

\[
\text{ind npluszero n =}
\]

\[
\text{case n [eq] of}
\]

\[
\text{Zero } \rightarrow \text{unfold (0 + 0) in _}
\]

\[
\text{Suc m } \rightarrow
\]

\[
\text{let _ = npluszero m [ord eq] in}
\]

\[
\text{unfold ((Suc m) + 0) in _}
\]

The expression \text{unfold a in b} expands to

\[
\text{let _ = (join : a = a1) in}
\]

\[
\text{let _ = (join : a1 = ...) in}
\]

\[
\text{...}
\]

\[
\text{let _ = (join : ... = an) in}
\]

\[
b
\]

\[
\text{when a } \rightsquigarrow a1 \rightsquigarrow \ldots \rightsquigarrow \text{an}
\]
Extension: Reduction Modulo

\[ \text{log npluszero : (n : Nat) \to (n + 0 = n)} \]
\[ \text{ind npluszero n =} \]
\[ \quad \text{case n [eq] of} \]
\[ \quad \quad \text{Zero } \rightarrow \text{ unfold (n + 0) in } _{\_} \]
\[ \quad \quad \text{Suc m } \rightarrow \]
\[ \quad \quad \quad \text{let ih = npluszero m [ord eq] in} \]
\[ \quad \quad \quad \text{unfold (n + 0) in } _{\_} \]

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

\[ n + 0 \sim_{\text{cbv}} 0 \]
Extension: Smart join

\[
\text{log npluszero} : (n : \text{Nat}) \rightarrow (n + 0 = n)
\]
\[
\text{ind npluszero} n =
\text{case } n \text{ [eq] of}
\]
\[
\text{Zero} \rightarrow \text{smartjoin}
\]
\[
\text{Suc } m \rightarrow
\text{let } \text{ih} = \text{npluszero } m \text{ [ord eq] in}
\text{smartjoin}
\]

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

\[
\text{snoc : List } \rightarrow \text{ A } \rightarrow \text{ List}
\]
\[
\text{snoc } xs \ x = xs ++ x :: []
\]

How would you prove the following property in Agda?

\[
\text{snoc-inv} : \forall \ xs \ ys \ z \rightarrow (\text{snoc } xs \ z \equiv \text{snoc } ys \ z) \rightarrow xs \equiv ys
\]
\[
\text{snoc-inv} (x :: xs') (y :: ys') z pf with (\text{snoc } xs' \ z) | (\text{snoc } ys' \ z)
\]
\[
| \text{inspect (snoc } xs' \ z) | \text{inspect (snoc } ys' \ z)
\]
\[
\text{snoc-inv} (.y :: xs') (y :: ys') z \text{ refl s s}
\]
\[
| [ p ] | [ q ] \text{ with (snoc-inv xs' ys' z (trans p (sym q)))}
\]
\[
\text{snoc-inv} (.y :: .ys') (y :: ys') z \text{ refl s s}
\]
\[
| [ p ] | [ q ] | \text{refl = refl}
\]

Uses Agda idiom called “inspect on steroids.”
Zombie solution is more straightforward:

```haskell
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 \texttt{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 $\beta$-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 $\beta$-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 $\beta$-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.