Combining Proofs and Programs in a Dependently Typed Language

Stephanie Weirich

University of Pennsylvania

July 7, 2014

Certification of High-level and Low-level Programs
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 other dependent type theories)

```haskell
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 other dependent type theories)

```
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 other dependent type theories)

```
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 $\theta = L | P$)

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

which in turn specifies the properties of the fragment.

**Theorem (Type Soundness)**

If $\cdot \vdash^\theta a : A$ and if $a \rightsquigarrow^* v$ then $\cdot \vdash^\theta v : A$

**Theorem (Consistency)**

If $\cdot \vdash^L a : A$ then $a \rightsquigarrow^* v$
Reasoning about programs

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

\[ \text{log div62 : div 6 2 = 3} \]
\[ \text{log div62 = join} \]

(Here \text{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.

\[ \text{log div62 : div 6 2 = 3} \]
\[ \text{log div62 = join} \]

(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 $\beta$

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

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

the expression $f \ x$ does *not* type check because `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

\[
  f : \text{Vec Nat} 3 \rightarrow \text{Nat} \\
  x : \text{Vec Nat} (\text{div} 6 2)
\]

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.

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

Yes.

```lean
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?

\[
x : a = b \in \Gamma \\
\frac{}{\Gamma \vdash a = b}
\]

\[
\Gamma \vdash a = b \\
\frac{}{\Gamma \vdash \{a/x\} c = \{b/x\} c}
\]

\[
\frac{\Gamma \vdash a = a}{\Gamma \vdash a = a}
\]

\[
\frac{\Gamma \vdash b = a}{\Gamma \vdash b = a}
\]

\[
\frac{\Gamma \vdash a = b}{\Gamma \vdash a = c}
\]

\[
\frac{\Gamma \vdash a = b}{\Gamma \vdash b = c}
\]

Efficient algorithms for deciding this relation exist [Nieuwenhuis and Oliveras, 2007].
But, extending this relation with $\beta$-conversion makes it undecidable.
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
```
How do we know this works?

- 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
How do we know this works?

- 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/
How do we know this works?

- 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.
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 erasable information (type annotations, type coercions, erasable arguments).
Propositional equality in core language

- Primitive type constructor $A = B$
Propositional equality in core language

- Primitive type constructor $A = B$
- Heterogeneous (two sides need not have the same type)
Propositional equality in core language

- Primitive type constructor $A = B$
- Heterogeneous (two sides need not have the same type)
- Ignores *erasable* parts of terms $a = |a|$
Propositional equality in core language

- Primitive type constructor $A = B$
- Heterogeneous (two sides need not have the same type)
- Ignores *erasable* parts of terms $a = |a|$
- Eliminated by type coercion in core language:

\[
\Gamma \vdash^\theta a : A \quad \Gamma \vdash^L b : A = B \quad \Gamma \vdash B : \text{Type}
\]

\[
\Gamma \vdash^\theta a_{\triangleright b} : B
\]
Propositional equality in core language

- Primitive type constructor $A = B$
- Heterogeneous (two sides need not have the same type)
- Ignores *erasable* parts of terms $a = |a|$
- Eliminated by type coercion in core language:

$$
\Gamma \vdash^\theta a : A \quad \Gamma \vdash^L b : A = B \quad \Gamma \vdash B : \text{Type}
$$

$$
\Gamma \vdash^\theta a_{\succ b} : B
$$

- Type coercion is erasable $a_{\succ b} = a$
Propositional equality in core language

- Primitive type constructor $A = B$
- Heterogeneous (two sides need not have the same type)
- Ignores *erasable* parts of terms $a = |a|$
- Eliminated by type coercion in core language:

$$
\frac{
\Gamma \vdash^\theta a : A \quad \Gamma \vdash^L b : A = B \quad \Gamma \vdash B : \text{Type}
}{
\Gamma \vdash^\theta a \triangleright b : B
}
$$

- Type coercion is erasable $a \triangleright b = a$
- Includes injectivity of type and data constructors

$$(x : A) \to B = (x : A') \to B' \text{ implies } A = A'$$
Congruence closure in ZOMBIE

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

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

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
Extensions and Examples
Proof inference

Congruence closure can supply proofs of equality

\[
\begin{align*}
\text{log } n\text{pluszero : } & (n : \text{Nat}) \rightarrow (n + 0 = n) \\
\text{ind } n\text{pluszero } n = & \\
& \text{case } n \text{ [eq] of} \\
& \text{Zero } \rightarrow \\
& \quad \text{let } j = (\text{join : } 0 + 0 = 0) \text{ in } _{} \\
& \text{Suc } m \rightarrow \\
& \quad \text{let } \text{ih} = n\text{pluszero } m \text{ [ord eq] in} \\
& \quad \text{let } k = (\text{join : } (\text{Suc } m) + 0 = \text{Suc } (m + 0)) \text{ in } _{}
\end{align*}
\]
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

\[
\begin{align*}
\log \ npluszero : (n : \text{Nat}) \rightarrow (n + 0 = n) \\
\text{ind } npluszero \ n = \\
\hspace{1em} \text{case } n \; [\text{eq}] \; \text{of} \\
\hspace{2em} \text{Zero } \rightarrow \text{unfold } (n + 0) \; \text{in } _- \\
\hspace{2em} \text{Suc } m \rightarrow \\
\hspace{3em} \text{let } \text{ih} = npluszero \ m \; [\text{ord eq}] \; \text{in} \\
\hspace{3em} \text{unfold } (n + 0) \; \text{in } _-
\end{align*}
\]

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

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

Use unfold (and reduction modulo) on both sides of an equality when type checking \text{join}.
Smart case
An Agda Puzzle

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

\[
\text{snoc} : \text{List} \to A \to \text{List} \\
\text{snoc} \; xs \; x = xs \; ++ \; (x :: [])
\]

How would you prove the following property in Agda?

\[
\text{snoc-inv} : \forall \; xs \; ys \; z \to (\text{snoc} \; xs \; z \equiv \text{snoc} \; ys \; z) \to xs \equiv ys \\
\text{snoc-inv} \; (x :: xs') \; (y :: ys') \; z \; \text{pf} = ?
\]

\[
\ldots
\]
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.
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.
Thanks!