Combining Proofs and Programs in Trellys

Stephanie Weirich

University of Pennsylvania

May 26, 2011

MFPS 27

The TRELLYS project

The TRELLYS project

Penn University of Pennsylvania	The University of Iowa	Portland State University
Stephanie Weirich	Aaron Stump	Tim Sheard
Chris Casinghino	Harley Eades	Ki Yung Ahn
Vilhelm Sjöberg	Peng (Frank) Fu	Nathan Collins

Garrin Kimmell

A collaborative project to design a statically-typed functional programming language based on dependent type theory.

The TRELLYS project

Penn University *of* Pennsylvania	The University of Iowa	Portland State University
Stephanie Weirich	Aaron Stump	Tim Sheard
Chris Casinghino	Harley Eades	Ki Yung Ahn
Vilhelm Sjöberg	Peng (Frank) Fu	Nathan Collins

Garrin Kimmell

A collaborative project to design a statically-typed functional programming language based on dependent type theory.

Work-in-progress

Why Dependent Types?

- *Lightweight verification*: Dependent types express application-specific program invariants that are beyond the scope of existing type systems. *Example: Trees that satisfy the binary-search tree invariant*

Why Dependent Types?

- *Lightweight verification*: Dependent types express application-specific program invariants that are beyond the scope of existing type systems. *Example: Trees that satisfy the binary-search tree invariant*
- *Expressiveness*: Dependent types enable flexible interfaces, allowing more programs to be statically checked. *Examples: metaprogramming, variable arity-polymorphism, type-directed programming.*

Why Dependent Types?

- *Lightweight verification*: Dependent types express application-specific program invariants that are beyond the scope of existing type systems. *Example: Trees that satisfy the binary-search tree invariant*
- *Expressiveness*: Dependent types enable flexible interfaces, allowing more programs to be statically checked. *Examples: metaprogramming, variable arity-polymorphism, type-directed programming.*
- *Uniformity*: Full-spectrum dependent types provide the same syntax and semantics for program computations, type-level computations, and proofs.

An incremental approach

Start with a general purpose, call-by-value, functional programming language and strengthen its type system.

- Want to reuse existing ideas from FP languages
- Want to draw programmers from FP communities
- Want existing code to work with minor modification

An incremental approach

Start with a general purpose, call-by-value, functional programming language and strengthen its type system.

- Want to reuse existing ideas from FP languages
- Want to draw programmers from FP communities
- Want existing code to work with minor modification
- Want to support incremental verification... only provide the strongest guarantees about the most critical code.

On the shoulders of giants

Not the first to propose programming with dependent types

- Agda, Epigram, Coq, Lego, Nuprl

On the shoulders of giants

Not the first to propose programming with dependent types

- Agda, Epigram, Coq, Lego, Nuprl

Not the first functional language to incorporate ideas from Type Theory

- GHC, Ur, Sage, ATS, Ωmega, DML

On the shoulders of giants

Not the first to propose programming with dependent types

- Agda, Epigram, Coq, Lego, Nuprl

Not the first functional language to incorporate ideas from Type Theory

- GHC, Ur, Sage, ATS, Ωmega, DML

Not the first to propose a full-spectrum functional programming language based on dependent types

- Guru, Cayenne, Cardelli “A Polymorphic λ-calculus with Type:Type”

Why call-by-value?

- Have to choose something. Want to include nontermination so the order of evaluation makes a difference

Why call-by-value?

- Have to choose something. Want to include nontermination so the order of evaluation makes a difference
- Good cost model. Programmers can better predict the running time and space usage of their programs

Why call-by-value?

- Have to choose something. Want to include nontermination so the order of evaluation makes a difference
- Good cost model. Programmers can better predict the running time and space usage of their programs
- Distinction between values and computations built into the language. Variables stand for values, not computations

A programming language, not a logic

Can't use Curry-Howard Isomorphism to interpret this language as a logic.

A programming language, not a logic

Can’t use Curry-Howard Isomorphism to interpret this language as a logic.
All types are inhabited.

A programming language, not a logic

Can't use Curry-Howard Isomorphism to interpret this language as a logic.
All types are inhabited.

A seeming contradiction

How can we have a full-spectrum, dependently-typed language based on an inconsistent logic?

Syntactic type soundness

Main property of typed programming languages is proven by an *elementary syntactic* argument and extends in a straightforward manner to modern language features (such as references, concurrency, exceptions, continuations, etc.)

Syntactic type soundness

Main property of typed programming languages is proven by an *elementary syntactic* argument and extends in a straightforward manner to modern language features (such as references, concurrency, exceptions, continuations, etc.)

Theorem (Syntactic type soundness)

If $\vdash a : A$ *then either* a *diverges, aborts, or* $a \rightsquigarrow^{*}_{\mathsf{cbv}} v$ *and* $\vdash v : A$.

Syntactic type soundness

Main property of typed programming languages is proven by an *elementary syntactic* argument and extends in a straightforward manner to modern language features (such as references, concurrency, exceptions, continuations, etc.)

Theorem (Syntactic type soundness)

If $\vdash a : A$ *then either* a *diverges, aborts, or* $a \rightsquigarrow^*_{\mathsf{cbv}} v$ *and* $\vdash v : A$*.*

Type soundness gives us a form of *partial correctness*

Partial correctness

Can give a logical interpretation for *values* based on partial correctness:

$$\vdash a : \Sigma x\!:\!\mathsf{Nat}.\mathit{even}\ x = \mathit{true}$$

If a terminates, then it *must* produce a pair of a natural number and a *proof* that the result is even.

Partial correctness

Can give a logical interpretation for *values* based on partial correctness:

$$\vdash a : \Sigma x{:}\mathsf{Nat}.\mathit{even}\ x = \mathit{true}$$

If a terminates, then it *must* produce a pair of a natural number and a *proof* that the result is even.
But, implications may be bogus.

$$\vdash a : \Sigma x{:}\mathsf{Nat}.(\mathit{even}\ x = \mathit{true}) \to (x = \mathsf{3})$$

Partial correctness

Can give a logical interpretation for *values* based on partial correctness:

$$\vdash a : \Sigma x{:}\mathsf{Nat}.\mathit{even}\ x = \mathit{true}$$

If a terminates, then it *must* produce a pair of a natural number and a *proof* that the result is even.
But, implications may be bogus.

$$\vdash a : \Sigma x{:}\mathsf{Nat}.(\mathit{even}\ x = \mathit{true}) \to (x = \mathsf{3})$$

Type soundness tells us that trying to use the implication in some other proof could cause the program to diverge or abort, but not “go wrong.”

Total correctness

Partial correctness is not enough

- Can't compile this language efficiently (have to run proofs)
- Users are willing to work harder for stronger guarantees for critical code

From partial correctness to total correctness

Plan for the rest of the talk:

- Part I: present a full-spectrum CBV language that satisfies type soundness only
- Part II: identify a "logical" sublanguage and discuss the interactions between the two parts

From partial correctness to total correctness

Plan for the rest of the talk:

- Part I: present a full-spectrum CBV language that satisfies type soundness only
- Part II: identify a “logical” sublanguage and discuss the interactions between the two parts

Not covered by this talk:

- How to make type checking decidable by adding annotations to the syntax
- How to make program development feasible by inferring annotations

Part I : A call-by-value programming language with dependent types

Uniform language

Types, terms, kinds defined using the same syntax

Syntax

$$
\begin{array}{rcl}
a, b, A, B & ::= & \star \mid \mathsf{Nat} \mid (x : A) \to B \\
& \mid & 0 \mid \mathsf{S}\ a \mid \mathsf{case}\ a\ \mathsf{of}\ \{0 \Rightarrow a_1 ; \mathsf{S}\ x \Rightarrow a_2\} \\
& \mid & x \mid \mathsf{rec}\ f\ x . a \mid a\ b \\
& \mid & \mathsf{abort}
\end{array}
$$

$$
\begin{array}{rcl}
v, u & ::= & \star \mid \mathsf{Nat} \mid (x : A) \to B \\
& \mid & 0 \mid \mathsf{S}\ v \\
& \mid & x \mid \mathsf{rec}\ f\ x . a
\end{array}
$$

Call-by-value operational semantics

$$\boxed{a \rightsquigarrow_{\mathsf{cbv}} b}$$

$$\frac{}{(\mathsf{rec}\, f\ x.a)\ v \rightsquigarrow_{\mathsf{cbv}} [v/x][\mathsf{rec}\, f\ x.a/f]a}$$

$$\frac{}{\mathsf{case}\ 0\ \mathsf{of}\ \{0 \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow_{\mathsf{cbv}} a_1}$$

$$\frac{}{\mathsf{case}\ (\mathsf{S}\ v)\ \mathsf{of}\ \{0 \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow_{\mathsf{cbv}} [v/x]a_2}$$

$$\frac{}{\mathcal{E}[\mathsf{abort}] \rightsquigarrow_{\mathsf{cbv}} \mathsf{abort}}$$

Example

Polymorphic application

$$app : (x : \star) \to (f : x \to x) \to (z : x) \to x$$
$$app = \lambda x.\lambda f.\lambda z.f\ z$$

$$app\ \mathsf{Nat}\ (\lambda x.x)\ \mathsf{0} \equiv \mathsf{0}$$

Use standard abbreviations:

- $\lambda x.a$ for $\mathsf{rec}\ f\ x.a$ when f is not free in a
- $A \to B$ for $(x : A) \to B$ when x is not free in B

Expressive example

$$
\begin{aligned}
zeroApp &= \lambda g.\lambda z.g \\
oneApp &= \lambda g.\lambda z.g\ z \\
twoApp &= \lambda g.\lambda z.g\ z\ z
\end{aligned}
$$

$$
\begin{aligned}
nApp = \mathsf{rec}\ f\ n.\ \mathsf{case}\ n\ \mathsf{of}& \\
\{\ \mathsf{0} &\Rightarrow \lambda g.\lambda z.g\ ; \\
\mathsf{S}\ m &\Rightarrow \lambda g.\lambda z.f\ m\ (g\ z)\ z\}
\end{aligned}
$$

Expressive example

$$
\begin{array}{lcl}
zeroApp & : & \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
zeroApp & = & \lambda g.\lambda z.g \\
oneApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
oneApp & = & \lambda g.\lambda z.g\ z \\
twoApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
twoApp & = & \lambda g.\lambda z.g\ z\ z
\end{array}
$$

$$
\begin{array}{l}
nApp = \mathsf{rec}\ f\ n.\ \mathsf{case}\ n\ \mathsf{of} \\
\qquad\qquad \{\ \mathsf{0} \quad \Rightarrow \quad \lambda g.\lambda z.g\ ; \\
\qquad\qquad\ \ \ \mathsf{S}\ m \quad \Rightarrow \quad \lambda g.\lambda z.f\ m\ (g\ z)\ z\}
\end{array}
$$

Expressive example

$$\begin{array}{lcl} zeroApp & : & \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\ zeroApp & = & \lambda g.\lambda z.g \\ oneApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\ oneApp & = & \lambda g.\lambda z.g\ z \\ twoApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\ twoApp & = & \lambda g.\lambda z.g\ z\ z \end{array}$$

$$\begin{array}{l} nApp\ :\ (n{:}\mathsf{Nat}) \rightarrow (N\ \ n) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\ nApp = \mathsf{rec}\ f\ n.\ \mathsf{case}\ n\ \mathsf{of} \\ \qquad\qquad \begin{array}{lcl} \{\ \mathsf{0} & \Rightarrow & \lambda g.\lambda z.g\ ; \\ \ \ \mathsf{S}\ m & \Rightarrow & \lambda g.\lambda z.f\ \ m\ (g\ \ z)\ z\} \end{array} \end{array}$$

Expressive example

$$\begin{array}{lcl}
zeroApp & : & \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
zeroApp & = & \lambda g.\lambda z.g \\
oneApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
oneApp & = & \lambda g.\lambda z.g\ z \\
twoApp & : & (\mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
twoApp & = & \lambda g.\lambda z.g\ z\ z
\end{array}$$

$$\begin{array}{lcl}
N & : & \mathsf{Nat} \rightarrow * \\
N & = & \mathsf{rec}\ f\ n.\ \mathsf{case}\ n\ \mathsf{of} \\
 & & \quad \{\ 0 \quad \Rightarrow \quad \mathsf{Nat}\ ; \\
 & & \quad\ \ \mathsf{S}\ m \quad \Rightarrow \quad \mathsf{Nat} \rightarrow f\ m\} \\
nApp & : & (n{:}\mathsf{Nat}) \rightarrow (N\ n) \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} \\
nApp & = & \mathsf{rec}\ f\ n.\ \mathsf{case}\ n\ \mathsf{of} \\
 & & \quad \{\ 0 \quad \Rightarrow \quad \lambda g.\lambda z.g\ ; \\
 & & \quad\ \ \mathsf{S}\ m \quad \Rightarrow \quad \lambda g.\lambda z.f\ m\ (g\ z)\ z\}
\end{array}$$

Typing relation

$\boxed{\Gamma \vdash a : A}$

General recursion

$$\frac{\Gamma, y : A, f : (y{:}A) \to B \vdash a : B}{\Gamma \vdash \mathsf{rec}\ f\ y.a : (y{:}A) \to B}$$

Explicit failure

$$\frac{\Gamma \vdash A : \star}{\Gamma \vdash \mathsf{abort} : A}$$

Type is a type

$$\frac{}{\vdash \star : \star}$$

Conversion

Because types depend on programs, we want to identify types that contain equivalent programs.

$$\mathsf{Vec\ Nat\ (1+2)} \equiv \mathsf{Vec\ Nat\ 3}$$

Expressions can be assigned any equivalent type

Conversion

$$\frac{\Gamma \vdash a : A \quad A \equiv B \quad \Gamma \vdash B : \star}{\Gamma \vdash a : B}$$

Conversion

Because types depend on programs, we want to identify types that contain equivalent programs.

$$\mathsf{Vec\ Nat}\ (1+2) \equiv \mathsf{Vec\ Nat}\ 3$$

Expressions can be assigned any equivalent type

Conversion

$$\frac{\Gamma \vdash a : A \quad A \equiv B \quad \Gamma \vdash B : \star}{\Gamma \vdash a : B}$$

But what does it mean for types to be equal?

Definitional Equality

- Based on operational semantics (hence undecidable)
- Ideally: identify all terms that are contextually equivalent to each other
- For now: close step relation under reflexivity, symmetry, transitivity and substitutivity
- Strictly computational

$\boxed{a \equiv b}$

$$\frac{a_1 \leadsto_{\mathsf{cbv}} a_2}{a_1 \equiv a_2} \qquad \frac{}{a \equiv a} \qquad \frac{a_1 \equiv a_2}{a_2 \equiv a_1}$$

$$\frac{a_1 \equiv a_2 \quad a_2 \equiv a_3}{a_1 \equiv a_3} \qquad \frac{a_1 \equiv a_2}{[a_1/x]A \equiv [a_2/x]A}$$

Internalizing equality

Internalize definitional equality as a proposition, with a trivial proof

$$a, b, A, B ::= \ldots \mid a = b \mid \mathsf{join}$$

Internalizing equality

Internalize definitional equality as a proposition, with a trivial proof

$$a, b, A, B ::= \ldots \mid a = b \mid \mathsf{join}$$

Trivial proof holds when terms are definitionally equal and the proposition is well-formed

$$\frac{a \;\equiv\; b \quad \Gamma \vdash a = b : \star}{\Gamma \vdash \mathsf{join} : a = b}$$

Internalizing equality

Internalize definitional equality as a proposition, with a trivial proof

$$a, b, A, B ::= \ldots \mid a = b \mid \mathsf{join}$$

Trivial proof holds when terms are definitionally equal and the proposition is well-formed

$$\frac{a \;\equiv\; b \quad \Gamma \vdash a = b : \star}{\Gamma \vdash \mathsf{join} : a = b}$$

Because definitional equality is untyped, propositional equality is heterogeneous

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

Conversion and propositional equality

Extend conversion rule to propositional equality

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B}{\Gamma \vdash a : B}$$

Conversion and propositional equality

Extend conversion rule to propositional equality

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B}{\Gamma \vdash a : B}$$

- Subsumes previous conversion rule (using join as the value)

Conversion and propositional equality

Extend conversion rule to propositional equality

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B}{\Gamma \vdash a : B}$$

- Subsumes previous conversion rule (using join as the value)
- Conversion is implicit. Terms that differ only in convertible types are trivially equal

Conversion and propositional equality

Extend conversion rule to propositional equality

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B}{\Gamma \vdash a : B}$$

- Subsumes previous conversion rule (using join as the value)
- Conversion is implicit. Terms that differ only in convertible types are trivially equal
- Proof must be a *value* because of partial correctness

Conversion and propositional equality

Extend conversion rule to propositional equality

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B}{\Gamma \vdash a : B}$$

- Subsumes previous conversion rule (using join as the value)
- Conversion is implicit. Terms that differ only in convertible types are trivially equal
- Proof must be a *value* because of partial correctness
- Don't care which value it is

Why can we do this?

Type soundness follows the following property (which can be proven *syntactically*):

Lemma (Soundness of propositional equality)

If $\vdash v : A_1 = A_2$ *then* $A_1 \equiv A_2$.

The cost of CBV

Call-by-value semantics adds extra hypothesis to application rule:

$$\frac{\Gamma \vdash a : (x : A) \to B \quad \Gamma \vdash b : A \quad \color{red}{\Gamma \vdash [b/x]B : \star}}{\Gamma \vdash a\ b : [b/x]B}$$

The cost of CBV

Call-by-value semantics adds extra hypothesis to application rule:

$$\frac{\Gamma \vdash a : (x : A) \to B \quad \Gamma \vdash b : A \quad \Gamma \vdash [b/x]B : \star}{\Gamma \vdash a\ b : [b/x]B}$$

If b is a non-value, the rule must make sure that x was never treated as a value in B.

Implicit arguments

Some values have no runtime effect.
Useful for:

- Parametric polymorphism $(x : \star) \rightarrow x \rightarrow x$
- Preconditions $(x : \mathsf{Nat}) \rightarrow \neg(x = \mathsf{0}) \rightarrow \mathsf{Nat}$

Want to elide them from the syntax of terms

$$\mathit{app}\ (\lambda x.x)\ \mathsf{0} \quad \text{instead of} \quad \mathit{app}\ \mathsf{Nat}\ (\lambda x.x)\ \mathsf{0}$$

Implicit arguments

Add implicit abstraction type

$$a, b, A, B ::= \ldots \mid [x : A] \to B$$

but... can only generalize over values

$$\frac{\Gamma, x : A \vdash v : B \quad x \notin \mathrm{FV}\, v}{\Gamma \vdash v : [x : A] \to B}$$

...can only instantiate with values

$$\frac{\Gamma \vdash a : [x : A] \to B \quad \Gamma \vdash v : A}{\Gamma \vdash a : [v/x]B}$$

Value restrictions are annoying

Suppose we write a program that *proves* the following fact about natural numbers:

$$f : (x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to (x = \mathsf{S}\ y) \to \neg(x = \mathsf{0})$$

However, a use of this lemma "$f\ x\ y\ z$" is not a value and cannot be erased.

Value restrictions are annoying

Suppose we write a program that *proves* the following fact about natural numbers:

$$f : (x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to (x = \mathsf{S}\ y) \to \neg(x = \mathsf{0})$$

However, a use of this lemma "$f\ x\ y\ z$" is not a value and cannot be erased.
Must first use an explicit argument to evaluate it to a value, even though the value is irrelevant.

Taking stock

- Make type checking decidable by adding annotations to the syntax
- Make program development feasible by inferring annotations

Taking stock

- Make type checking decidable by adding annotations to the syntax
- Make program development feasible by inferring annotations
- ...but, irrelevant computations remain at runtime

Taking stock

- Make type checking decidable by adding annotations to the syntax
- Make program development feasible by inferring annotations
- ...but, irrelevant computations remain at runtime
- ...slowing execution

Taking stock

- Make type checking decidable by adding annotations to the syntax
- Make program development feasible by inferring annotations
- ...but, irrelevant computations remain at runtime
- ...slowing execution
- ...weakening equivalence

Taking stock

- Make type checking decidable by adding annotations to the syntax
- Make program development feasible by inferring annotations
- ...but, irrelevant computations remain at runtime
- ...slowing execution
- ...weakening equivalence
- ...and weakening static guarantees

Part II : A logical sublanguage

A logical language

- There is a logically-consistent sublanguage hiding in here.

A logical language

- There is a logically-consistent sublanguage hiding in here.
- How do we identify it?

A logical language

- There is a logically-consistent sublanguage hiding in here.
- How do we identify it?
- We use the type system!

A logical language

- There is a logically-consistent sublanguage hiding in here.
- How do we identify it?
- We use the type system!
- Annotate typing judgement to specify the *logical* language or the *programmatic* language.

A logical language

- There is a logically-consistent sublanguage hiding in here.
- How do we identify it?
- We use the type system!
- Annotate typing judgement to specify the *logical* language or the *programmatic* language.

New typing judgement form:

$$\Gamma \vdash^{\theta} a : A \quad \textit{where} \quad \theta ::= \mathsf{L} \mid \mathsf{P}$$

Subsumption

Logical language is a *sublanguage* of the programmatic language.

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}$$

It guarantees stronger properties about its expressions.

Theorem (Syntactic type soundness)

If $\vdash^{\mathsf{P}} a : A$ *then either* a *diverges, aborts, or* $a \leadsto^{*}_{\mathsf{cbv}} v$ *and* $\vdash^{\mathsf{P}} v : A$.

Theorem (Semantic consistency)

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

Expressive features must be programmatic

Some capabilities only available for the programmatic language

Type-In-Type

$$\frac{}{\vdash^{\mathsf{P}} \star : \star}$$

Failure

$$\frac{\Gamma \vdash^{\mathsf{P}} A : \star}{\Gamma \vdash^{\mathsf{P}} \mathsf{abort} : A}$$

General recursion

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{P}} (x :^{\theta} A) \to B : \star \\ \Gamma, x :^{\theta} A, f :^{\mathsf{P}} (x :^{\theta} A) \to B \vdash^{\mathsf{P}} b : B\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\ x.b : (x :^{\theta} A) \to B}$$

What does the logical language look like?

Logical functions should not be recursive...

$$\frac{\mathsf{\Gamma} \vdash^{\mathsf{L}} (x :^{\theta} A) \to\ B : \star \quad \mathsf{\Gamma}, x :^{\theta} A \vdash^{\mathsf{L}} b : B}{\mathsf{\Gamma} \vdash^{\mathsf{L}} \mathsf{rec}\, f\ x.b : (x :^{\theta} A) \to\ B}$$

What does the logical language look like?

Logical functions should not be recursive...

$$\frac{\Gamma \vdash^{\mathsf{L}} (x :^{\theta} A) \to\ B : \star \quad \Gamma, x :^{\theta} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} \mathsf{rec}\, f\ x.b : (x :^{\theta} A) \to\ B}$$

...except for primitive recursion over natural numbers

$$\frac{\begin{array}{l}\Gamma, x :^{\mathsf{L}} \mathsf{Nat} \vdash^{\mathsf{L}} B : \star \\ \Gamma, x :^{\mathsf{L}} \mathsf{Nat}, f :^{\mathsf{L}} (y :^{\mathsf{L}} \mathsf{Nat}) \to\ [z :^{\mathsf{L}} (\mathsf{S}\ y) = x] \to [y/x]B \vdash^{\mathsf{L}} b : B\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{rec}\, f\ x.b : (x :^{\mathsf{L}} \mathsf{Nat}) \to\ B}$$

Mixing the sublanguages

Programmatic functions can have logical parameters:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{P}} (x :^{\mathsf{L}} A) \to B : \star \\ \Gamma, x :^{\mathsf{L}} A, f :^{\mathsf{P}} (x :^{\mathsf{L}} A) \to B \vdash^{\mathsf{P}} b : B\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\ x.b : (x :^{\mathsf{L}} A) \to B}$$

Such arguments are logical “proofs” that the preconditions of the function are satisfied.

Mixing the sublanguages

Programmatic functions can have logical parameters:

$$\frac{\begin{array}{c}\mathsf{\Gamma} \vdash^{\mathsf{P}} (x :^{\mathsf{L}} A) \to\ B : \star \\ \mathsf{\Gamma}, x :^{\mathsf{L}} A, f :^{\mathsf{P}} (x :^{\mathsf{L}} A) \to\ B \vdash^{\mathsf{P}} b : B\end{array}}{\mathsf{\Gamma} \vdash^{\mathsf{P}} \mathsf{rec}\, f\ x.b : (x :^{\mathsf{L}} A) \to\ B}$$

Such arguments are logical “proofs” that the preconditions of the function are satisfied.

These arguments can be implicit, even if they are not values.

Freedom of Speech

Logical functions can have programmatic parameters:

$$\frac{\Gamma \vdash^{\mathsf{L}} (x :^{\mathsf{P}} A) \to B : \star \quad \Gamma, x :^{\mathsf{P}} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} \mathsf{rec}\, f\ x.b : (x :^{\mathsf{P}} A) \to B}$$

Freedom of Speech

Logical functions can have programmatic parameters:

$$\frac{\Gamma \vdash^{\mathsf{L}} (x :^{\mathsf{P}} A) \to B : \star \quad \Gamma, x :^{\mathsf{P}} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} \mathsf{rec}\, f\ x.b : (x :^{\mathsf{P}} A) \to B}$$

Application restricted to terminating arguments.

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{L}} a : (x :^{\mathsf{P}} A) \to B \\ \Gamma \vdash_{\downarrow} b : A \quad \Gamma \vdash^{\mathsf{L}} [b/x]B : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} a\ b : [b/x]B}$$

Total arguments are either logical or values.

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash_{\downarrow} a : A} \qquad \frac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash_{\downarrow} v : A}$$

Conversion

- Conversion available for both languages
- Equality proof must be total

$$\frac{\Gamma \vdash^{\theta} a : A \quad \Gamma \vdash_{\downarrow} b : A = B}{\Gamma \vdash^{\theta} a : B}$$

Shared values

Some values are shared between the two languages.

Shared values

Some values are shared between the two languages.
For example, all natural numbers are values in the logical language as well as in the programmatic language.

$$\frac{}{\vdash^{\mathsf{L}} \mathsf{Nat} : \star} \qquad \frac{}{\vdash^{\mathsf{L}} \mathsf{0} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash^{\theta} n : \mathsf{Nat}}{\Gamma \vdash^{\theta} \mathsf{S}\ n : \mathsf{Nat}}$$

Shared values

Some values are shared between the two languages.
For example, all natural numbers are values in the logical language as well as in the programmatic language.

$$\frac{}{\vdash^{\mathsf{L}} \mathsf{Nat} : \star} \qquad \frac{}{\vdash^{\mathsf{L}} \mathsf{0} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash^{\theta} n : \mathsf{Nat}}{\Gamma \vdash^{\theta} \mathsf{S}\ n : \mathsf{Nat}}$$

This means that it is sound to treat a variable of type Nat as logical, no matter what it is assumed to be in the context.

$$\frac{\Gamma \vdash^{\mathsf{P}} x : \mathsf{Nat}}{\Gamma \vdash^{\mathsf{L}} x : \mathsf{Nat}}$$

Uniform equality

Equality proofs are also shared.
All equality proofs and propositions are logical, no matter what sort of terms they equate.

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \qquad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \star} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a = b : \star \qquad a \equiv b}{\Gamma \vdash^{\mathsf{L}} \mathsf{join} : a = b}$$

Uniform equality

Equality proofs are also shared.
All equality proofs and propositions are logical, no matter what sort of terms they equate.

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \qquad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \star} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a = b : \star \qquad a \equiv b}{\Gamma \vdash^{\mathsf{L}} \mathsf{join} : a = b}$$

We can treat a programmatic variable as a logical equality proof.

$$\frac{\Gamma \vdash^{\mathsf{P}} x : A = B}{\Gamma \vdash^{\mathsf{L}} x : A = B}$$

This supports incremental verification. We can have a partial function return an equality proof, and then use that to satisfy the preconditions of any part of the code.

Conclusion

Future work

- What logical system should we use? Predicative? Impredicative? Large Eliminations? Induction-Recursion?

Future work

- What logical system should we use? Predicative? Impredicative? Large Eliminations? Induction-Recursion?
- Interaction with classical reasoning: allow proofs to branch on whether a program halts, aborts or diverges

Future work

- What logical system should we use? Predicative? Impredicative? Large Eliminations? Induction-Recursion?
- Interaction with classical reasoning: allow proofs to branch on whether a program halts, aborts or diverges
- Strengthen definitional and propositional equality

Future work

- What logical system should we use? Predicative? Impredicative? Large Eliminations? Induction-Recursion?
- Interaction with classical reasoning: allow proofs to branch on whether a program halts, aborts or diverges
- Strengthen definitional and propositional equality
- Elaboration to an annotated language

Summary

- Can have full-spectrum dependently-typed language with nontermination, effects, etc.
- Call-by-value semantics permits “partial correctness”
- Logical and programmatic languages can interact
    - All proofs are programs
    - Logic can talk about programs
    - Shared values can be passed from programs to the logic