Mechanizing a Decision Procedure for Coproduct Equality in Twelf

Arbob Ahmad (applying to PhD programs this fall!)

Dan Licata

Robert Harper

Carnegie Mellon University

Equality

Equality of program terms is one of the thorniest issues in type theory. Various applications, including:

- justifying compiler optimizations
- dependency: program equality induces type equality

Equality

Equality of program terms is one of the thorniest issues in type theory. Various applications, including:

- justifying compiler optimizations
- dependency: program equality induces type equality

Thorny because you often want equality to be

- as coarse as possible
- decidable

Equality

STLC with finite products and sums:

$$\begin{array}{lll} \tau & ::= & \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \\ e & ::= & \ldots \mid \textsf{inl}\ e \mid \textsf{inr}\ e \mid \textsf{case}(e, x_1.e_1, x_2.e_2) \mid \ldots \end{array}$$

Equality: congruence $\Gamma \vdash e \equiv e' : \tau$

Equality

STLC with finite products and sums:

$$\begin{array}{lll} \tau & ::= & \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \\ e & ::= & \ldots \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}(e, x_1.e_1, x_2.e_2) \mid \ldots \end{array}$$

Equality: congruence $\Gamma \vdash e \equiv e' : \tau$

How is equality defined?

Equality for Sums

Standard $\beta\eta$ rules for $\to$, $\times$, 1 capture categorical universal properties

What is the equivalent for sums?

$$\frac{}{\mathsf{case}(\mathsf{inl}\ e, x_1 . e_1 , x_2 . e_2) \equiv [e/x] e_1}\ \beta_{+l}$$

$$\frac{}{\mathsf{case}(\mathsf{inr}\ e, x_1 . e_1 , x_2 . e_2) \equiv [e/x] e_2}\ \beta_{+r}$$

Equality for Sums

η for sums: “defines the same decision tree”

Pick any subterm of sum type and pivot on it at the outside:

$$\frac{e : \tau_1 + \tau_2}{[e/x]e' \equiv \mathsf{case}(e, x_1.[\mathsf{inl}\ x_1/x]e', x_2.[\mathsf{inr}\ x_2/x]e')}\ \eta_+$$

C.f. the uniqueness condition of the categorical universal property for coproducts.

Equality for Sums: Consequences

Reconstruction:

$$\frac{e : \tau_1 + \tau_2}{e \equiv \mathsf{case}(e, x_1.\mathsf{inl}\ x_1, x_2.\mathsf{inr}\ x_2)}$$

Equality for Sums: Consequences

Reconstruction:

$$\frac{e : \tau_1 + \tau_2}{e \equiv \mathsf{case}(e, x_1.\mathsf{inl}\ x_1, x_2.\mathsf{inr}\ x_2)}$$

Commuting conversions:

$$\frac{}{(\mathsf{case}(e, x_1.e_1, x_2.e_2))\ e' \equiv \mathsf{case}(e, x_1.e_1\ e', x_2.e_2\ e')}$$

Equality for Sums: Consequences

Reconstruction:

$$\frac{e : \tau_1 + \tau_2}{e \equiv \mathsf{case}(e, x_1.\mathsf{inl}\ x_1, x_2.\mathsf{inr}\ x_2)}$$

Commuting conversions:

$$\frac{}{(\mathsf{case}(e, x_1.e_1, x_2.e_2))\ e' \equiv \mathsf{case}(e, x_1.e_1\ e', x_2.e_2\ e')}$$

Permuting conversion (c.f. BDD variable ordering):

$\mathsf{case}(e, x_1.\mathsf{case}(f, y_1.f_1, y_2.f_2), x_2.e_2) \equiv$
$\mathsf{case}(f, y_1.\mathsf{case}(e, x_1.f_1, x_2.e_2), y_2.\mathsf{case}(e, x_1.f_2, x_2.e_2))$

Deciding Coproduct Equality is Tricky

η_+ has a very non-local flavor: pick *any* subterm and pivot on it

Previous decision procedures:

- Ghani (1995), Lindley (2007) using rewriting
- Altenkirch et al. (2001), Balat et al. (2004) using NBE/TDPE

This Work

- Give a new decision procedure for coproduct equality based on a *canonical forms* technique

- Mechanize the proof of its correctness in Twelf

This Work

- **Give a new decision procedure for coproduct equality based on a canonical forms technique**
- Mechanize the proof of its correctness in Twelf

Our Decision Procedure

To prove: $\Gamma \vdash e \equiv e' : \tau$ is decidable

Our approach: give a sound, complete, and decidable algorithmic definition of equality

1. Translate terms into a language of *canonical forms*. Many equal terms have the same canonical form
2. Compare canonical forms with a structural congruence to get the remaining equalities

Canonical Forms

Goal: only one way to write equivalent terms.
You can't write:

$\beta_{\rightarrow}$ $(\lambda\, x.\, x)\; y$, only y

β_{+} $\text{case}(\text{inl}\; x, x_1.\text{inl}\; x_1, x_2.())$, only $\text{inl}\; x$

η_1 x at type 1, only $()$

$\eta_{\rightarrow,1}$ f at type $1 \rightarrow 1$, only $\lambda\, \_.\, f\; ()$

Canonical Forms

Goal: only one way to write equivalent terms.
You can't write:

$\beta_{\rightarrow}$ $(\lambda x.\, x)\; y$, only y

β_{+} $\mathsf{case}(\mathsf{inl}\ x, x_1.\mathsf{inl}\ x_1, x_2.())$, only $\mathsf{inl}\ x$

η_{1} x at type 1, only $()$

$\eta_{\rightarrow,1}$ f at type $1 \rightarrow 1$, only $\lambda \_.\, f\ ()$

How do canonical forms control η_{+}?

Canonical Forms

- Monadic language based on CLF [Watkins et al., ’02]
- Distinction between asynchronous and synchronous types based on focussing [Andreoli ’92]

$$\begin{array}{lcl} A & ::= & A_1 \to A_2 \mid A_1 \times A_2 \mid 1 \mid \{S\} \\ S & ::= & A \mid S_1 + S_2 \mid 0 \end{array}$$

  Intuition:

  ▷ elims for synchronous types involve a third party

  ▷ monad controls the use of these elims

Canonical Forms

Consider $or : bool \rightarrow bool \rightarrow bool$. Two implementations:

1. λ`x.if x then `λ`_.true else `λ`y.y`
2. λ`x.`λ`y.if x then true else y`

Canonical Forms

Consider $or : bool \to bool \to bool$. Two implementations:

1. `λx.if x then λ_.true else λy.y`
2. `λx.λy.if x then true else y`

In canonical forms

- $bool \to (bool \to bool)$ is $\{bool\} \to (\{bool\} \to \{bool\})$
- Cannot write (1): case-analysis only when producing something of type $\{S\}$

Structural Congruence

Canonical forms don't get rid of all instances of η_+:

- Permuting conversion:
  $\mathsf{case}(e, x_1.\mathsf{case}(f, y_1.f_1, y_2.f_2), x_2.e_2)$ and
  $\mathsf{case}(f, y_1.\mathsf{case}(e, x_1.f_1, x_2.e_2), y_2.\mathsf{case}(e, x_1.f_2, x_2.e_2))$

- Dead code: $\mathsf{case}(e, x_1.(), x_2.\mathsf{case}(e, x_1.dead, x_2.()))$ and $\mathsf{case}(e, x_1.(), x_2.())$

- Indifference: $\mathsf{case}(e, x_1.(), x_2.())$ and $()$

Structural Congruence

Consequently, we compare canonical forms up to permuting conversions, dead code, and indifference.

Why bundle the three together?

- Permuting conversions are inherently symmetric, so neither side is to be preferred
- Permuting creates dead code and indifference

Translation to Canonical Forms

Need meta-operations to witness expected principles:

- Hereditary substitution: compute the canonical result of substituting one canonical form into another (witnesses cut admissibility)
- Expansion: expand a variable into a canonical term (witnesses identity principle)
- Inversion: rearrange a decision tree so that a specified term is case-analyzed first (witnesses coproduct reasoning)

Translation to Canonical Forms

Need meta-operations to witness expected principles:

- Hereditary substitution: compute the canonical result of substituting one canonical form into another (witnesses cut admissibility)
- Expansion: expand a variable into a canonical term (witnesses identity principle)
- Inversion: rearrange a decision tree so that a specified term is case-analyzed first (witnesses coproduct reasoning)

Translation from STLC to canonical forms is a simple outer induction using these judgements

Properties

- Totality: translation is a type-correct function
- Completeness: if two terms are equal, then they translate to congruent canonical forms
- Soundness: if two terms translate to congruent canonical forms, then they're equal
- Decidability: congruence of canonical forms is decidable

Properties

OK	Totality: translation is a type-correct function
Almost	Completeness: if two terms are equal, then they translate to congruent canonical forms [everything but functionality of hered. subst.]
To do	Soundness: if two terms translate to congruent canonical forms, then they’re equal
To do	Decidability: congruence of canonical forms is decidable

This Work

- Give a new decision procedure for coproduct equality based on a canonical forms technique
- **Mechanize the proof of its correctness in Twelf**

Syntax = LF Types and Constants

$$
\begin{array}{lcl}
A & ::= & A_1 \rightarrow A_2 \mid A_1 \times A_2 \mid 1 \mid \{S\} \\
S & ::= & A \mid S_1 + S_2 \mid 0
\end{array}
$$

```
atp : type.
stp : type.
```

```
arrow : atp -> atp -> atp.
prod  : atp -> atp -> atp.
one   : atp.
circ  : stp -> atp.
...
```

Syntax = LF Types and Constants

Terms ($\rightarrow$ and $+$ fragment):

LF type			Syntactic Category
`rtm`	R	::=	$x \mid R\ N$
`ntm`	N	::=	$\lambda\, x.\, N \mid \{E\}$
`etm`	E	::=	$M \mid R \gg I$
`mtm`	M	::=	$N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M$
`itm`	I	::=	$\mathsf{case}(I_1, I_2) \mid \mathsf{asynch}(x.E)$

Judgements = Indexed Type Families

Hereditary substitution $[N/x]_A R = N' : A'$

represented by

```
hsubst-rr : ntm -> atp -> (rtm -> rtm)
                -> ntm -> atp
                -> type.
```

Inference Rules = Constants

$$\frac{\begin{array}{l}[N_0/x]_{A_0} R_1 = \lambda\, y.\, N' : A_2 \to A \\ [N_0/x]_{A_0} N_2 = N_2' \\ [N_2'/y]_{A_2} N' = N''\end{array}}{[N_0/x]_{A_0} R_1\ N_2 = N'' : A}$$

```
c : hsubst-rr N0 A0 ([x] app (R1 x) (N2 x)) N″ A
     <- hsubst-rr N0 A0 R1 (lam N′) (arrow A2 A)
     <- hsubst-n N0 A0 N2 N2′
     <- hsubst-n N2′ A2 ([y] N′) N″.
```

Twelf Proves Termination

```
hsubst-rr : ntm -> atp -> (rtm -> rtm)
            -> ntm -> atp
            -> type.

%reduces A’ <= A0 (hsubst-rr _ A0 _ _ A’).
%worlds (x:rtm)* (hsubst-rr _ _ _ _ _) ... .
%terminates {(A0 ...) (R ...)}
       (hsubst-rr _ A0 R _ _) ... .
```

(Ellipses: mutually recursive with hsubst for the other syntactic categories)

Thm. Statements = Annotated Type Families

Theorem: If $\Gamma, y \Rightarrow A_2 \vdash N \Leftarrow A$ and $\Gamma \vdash N_2 \Leftarrow A_2$
then $[N_2/y]_{A_2} N = N'$ and $\Gamma \vdash N' \Leftarrow A$.

```
thm : {A2}
    ({y : rtm} {dy : synth y A2} ncheck (N y) A)
    -> ncheck N2 A2
    -> hsubst-n N2 A2 ([y] N y) N'
    -> ncheck N' A
    -> type.
%mode thm +A2 +D1 +D2 -D3 -D4.
%worlds (x:tm, dx: synth x A)* (thm _ _ _ _ _).
```

Proofs = Constants + Totality Check

```
l : thm A2
   ([x] [dx] (ncheck-lam ([y] [dy] DcN x dx y dy))
   DcN2

   (hsubst-n-lam DsN)
   (ncheck-lam DcN')

   <- ({y} {dy : synth y Af}
      thm A2 ([x] [dx] (DcN x dx y dy)) DcN2
         (DsN y) (DcN' y dy)).

%total {(A ...) (D ...)} (thm A D _ _ _) ... .
```

Proof So Far

(768) Represent syntax and judgements

(5558) Functionality and type-correctness of translation to canonical forms

(7617) Several lemmas leading up to completeness. E.g. both sides of

$$\frac{e : \tau_1 + \tau_2}{e \equiv \mathsf{case}(e, x_1.\mathsf{inl}\ x_1, x_2.\mathsf{inr}\ x_2)}$$

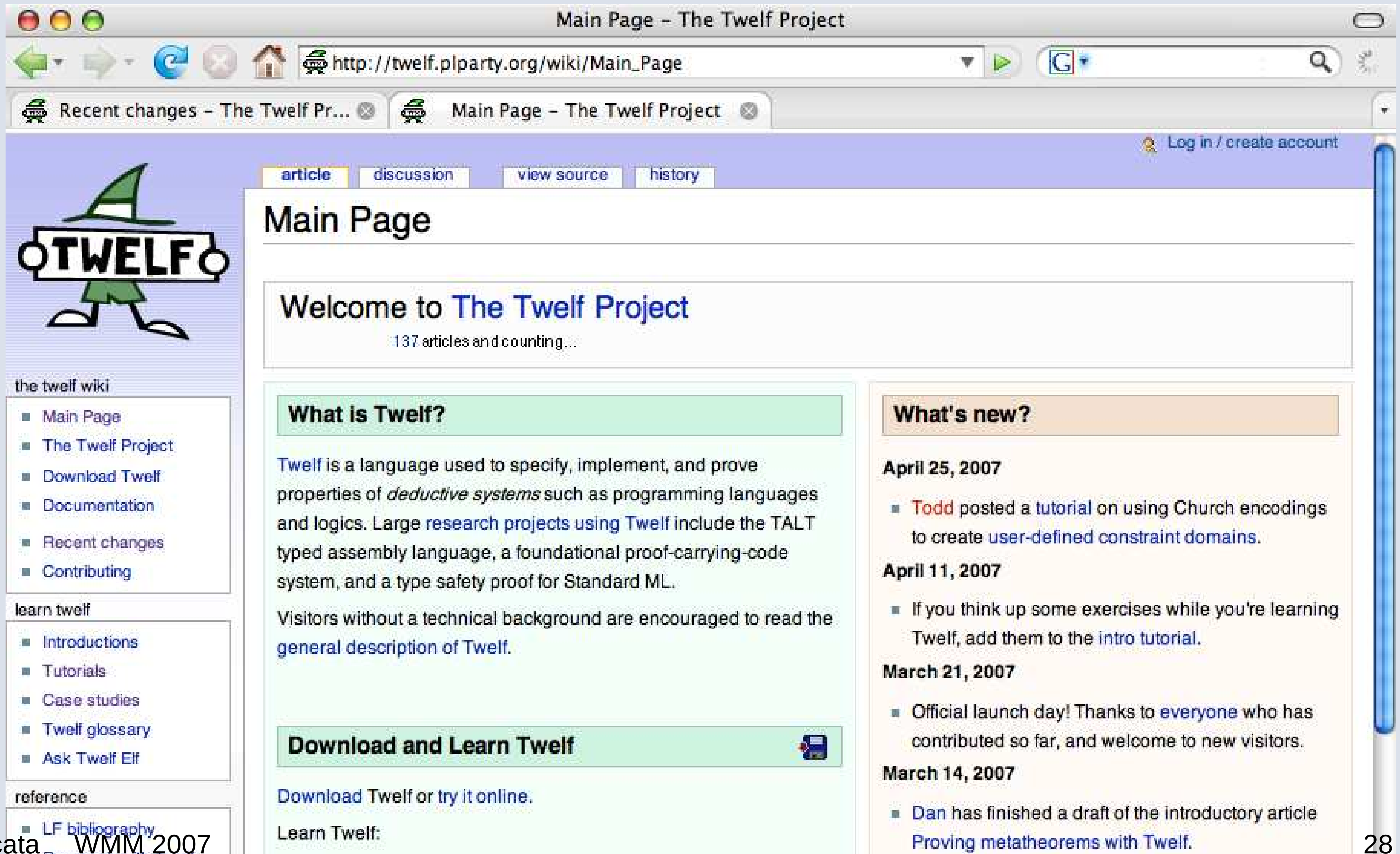
have the same canonical form

Proof Techniques

Uses lots of Twelf techniques:

- using `%reduces` for termination
- mutual induction and lexicographic induction
- reasoning with equality and respects lemmas
- proving uniqueness lemmas
- output factoring
- reasoning from false
- catch-all cases

http://twelf.plparty.org



Main Page – The Twelf Project

http://twelf.plparty.org/wiki/Main_Page

Recent changes – The Twelf Pr... | Main Page – The Twelf Project

Log in / create account

article | discussion | view source | history

the twelf wiki
- Main Page
- The Twelf Project
- Download Twelf
- Documentation
- Recent changes
- Contributing

learn twelf
- Introductions
- Tutorials
- Case studies
- Twelf glossary
- Ask Twelf Elf

reference
- LF bibliography

Main Page

Welcome to The Twelf Project

137 articles and counting...

What is Twelf?

Twelf is a language used to specify, implement, and prove properties of *deductive systems* such as programming languages and logics. Large research projects using Twelf include the TALT typed assembly language, a foundational proof-carrying-code system, and a type safety proof for Standard ML.

Visitors without a technical background are encouraged to read the general description of Twelf.

Download and Learn Twelf

Download Twelf or try it online.

Learn Twelf:

What's new?

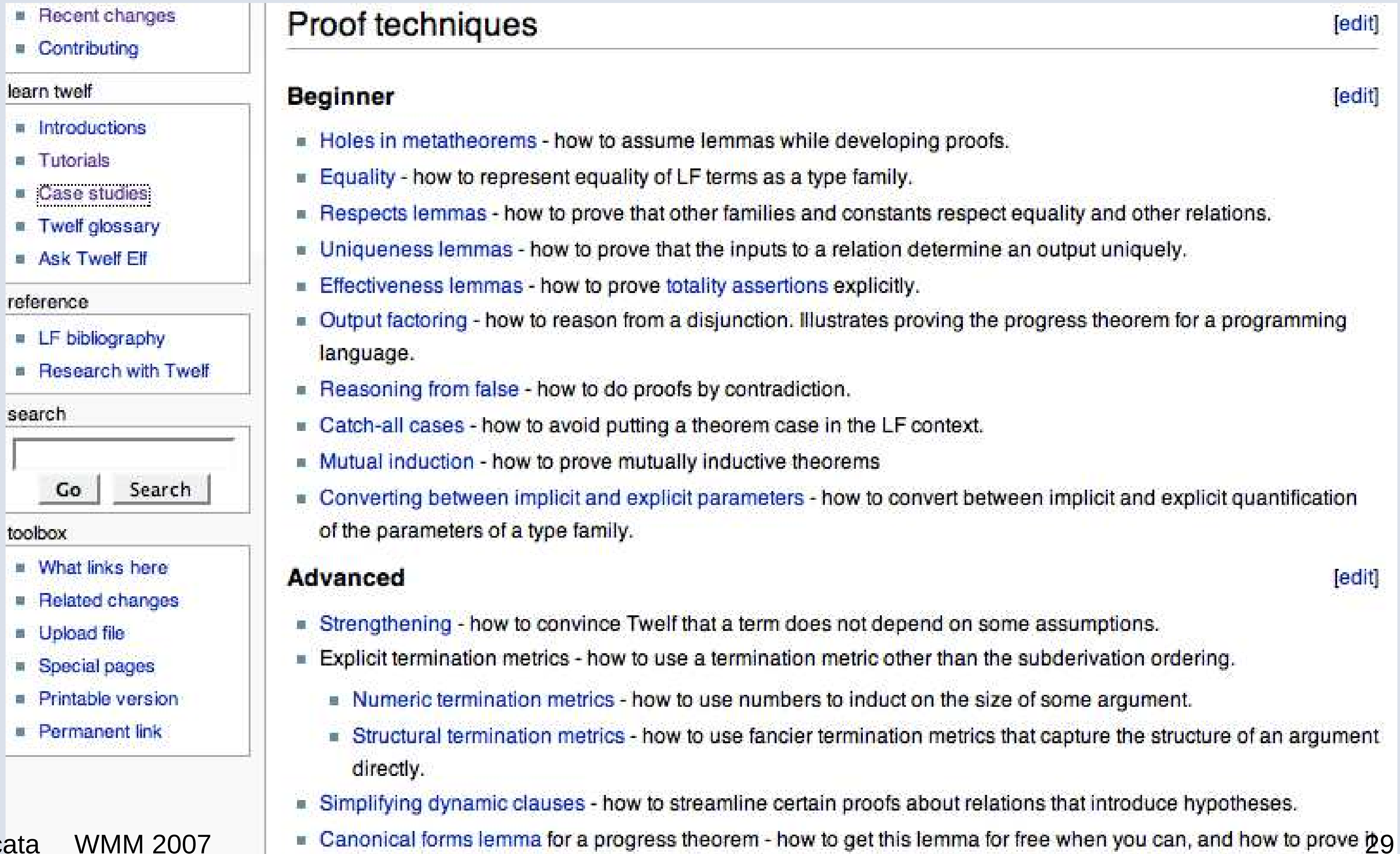
April 25, 2007
- Todd posted a tutorial on using Church encodings to create user-defined constraint domains.

April 11, 2007
- If you think up some exercises while you're learning Twelf, add them to the intro tutorial.

March 21, 2007
- Official launch day! Thanks to everyone who has contributed so far, and welcome to new visitors.

March 14, 2007
- Dan has finished a draft of the introductory article Proving metatheorems with Twelf.

http://twelf.plparty.org



- Recent changes
- Contributing

learn twelf

- Introductions
- Tutorials
- Case studies
- Twelf glossary
- Ask Twelf Elf

reference

- LF bibliography
- Research with Twelf

search

Go Search

toolbox

- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link

Proof techniques [edit]

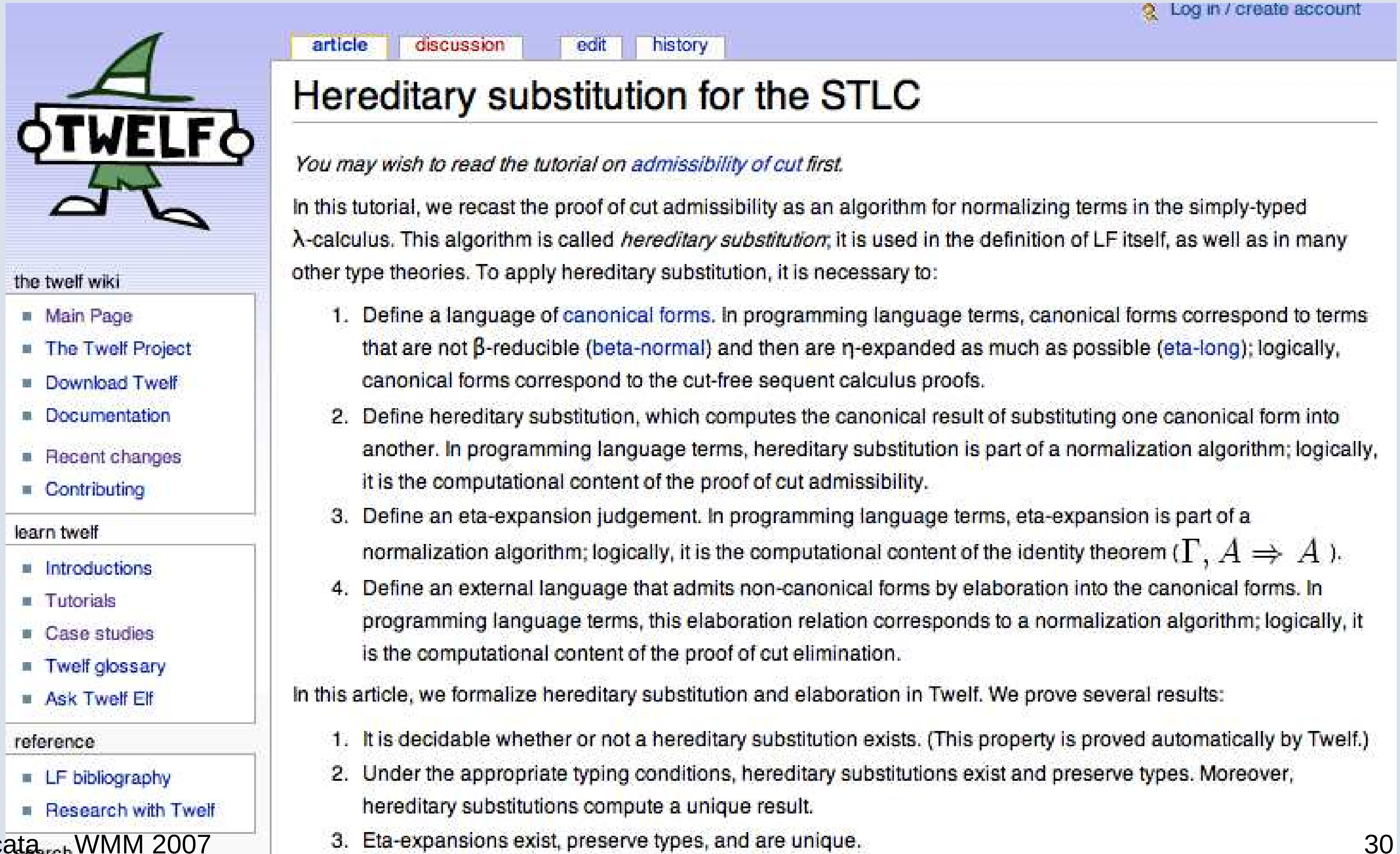
Beginner [edit]

- Holes in metatheorems - how to assume lemmas while developing proofs.
- Equality - how to represent equality of LF terms as a type family.
- Respects lemmas - how to prove that other families and constants respect equality and other relations.
- Uniqueness lemmas - how to prove that the inputs to a relation determine an output uniquely.
- Effectiveness lemmas - how to prove totality assertions explicitly.
- Output factoring - how to reason from a disjunction. Illustrates proving the progress theorem for a programming language.
- Reasoning from false - how to do proofs by contradiction.
- Catch-all cases - how to avoid putting a theorem case in the LF context.
- Mutual induction - how to prove mutually inductive theorems
- Converting between implicit and explicit parameters - how to convert between implicit and explicit quantification of the parameters of a type family.

Advanced [edit]

- Strengthening - how to convince Twelf that a term does not depend on some assumptions.
- Explicit termination metrics - how to use a termination metric other than the subderivation ordering.
  - Numeric termination metrics - how to use numbers to induct on the size of some argument.
  - Structural termination metrics - how to use fancier termination metrics that capture the structure of an argument directly.
- Simplifying dynamic clauses - how to streamline certain proofs about relations that introduce hypotheses.
- Canonical forms lemma for a progress theorem - how to get this lemma for free when you can, and how to prove it

http://twelf.plparty.org


Log in / create account

article discussion edit history

the twelf wiki

- Main Page
- The Twelf Project
- Download Twelf
- Documentation
- Recent changes
- Contributing

learn twelf

- Introductions
- Tutorials
- Case studies
- Twelf glossary
- Ask Twelf Elf

reference

- LF bibliography
- Research with Twelf

Hereditary substitution for the STLC

You may wish to read the tutorial on admissibility of cut first.

In this tutorial, we recast the proof of cut admissibility as an algorithm for normalizing terms in the simply-typed λ-calculus. This algorithm is called *hereditary substitution*; it is used in the definition of LF itself, as well as in many other type theories. To apply hereditary substitution, it is necessary to:

1. Define a language of canonical forms. In programming language terms, canonical forms correspond to terms that are not β-reducible (beta-normal) and then are η-expanded as much as possible (eta-long); logically, canonical forms correspond to the cut-free sequent calculus proofs.
2. Define hereditary substitution, which computes the canonical result of substituting one canonical form into another. In programming language terms, hereditary substitution is part of a normalization algorithm; logically, it is the computational content of the proof of cut admissibility.
3. Define an eta-expansion judgement. In programming language terms, eta-expansion is part of a normalization algorithm; logically, it is the computational content of the identity theorem ($\Gamma, A \Rightarrow A$).
4. Define an external language that admits non-canonical forms by elaboration into the canonical forms. In programming language terms, this elaboration relation corresponds to a normalization algorithm; logically, it is the computational content of the proof of cut elimination.

In this article, we formalize hereditary substitution and elaboration in Twelf. We prove several results:

1. It is decidable whether or not a hereditary substitution exists. (This property is proved automatically by Twelf.)
2. Under the appropriate typing conditions, hereditary substitutions exist and preserve types. Moreover, hereditary substitutions compute a unique result.
3. Eta-expansions exist, preserve types, and are unique.

Arbob’s Experience

“Determining the Twelf representation for the syntax and judgments was generally straightforward. Typically, the correct mechanization could be seen by direct analogy to some similar construct that appeared in an example on the wiki. In one case, the process of mechanizing the syntax and judgments actually revealed a superfluous term in our language, which we were then able to eliminate.”

Arbob’s Experience

“Of course, mechanizing the proofs was more challenging. Often when a proof seemed difficult to formalize or I was uncertain which lemmas the Twelf proof would require, there was a case study on the wiki that described the mechanization of a similar proof. The mechanized proofs themselves generally resembled my paper proofs. Typically, they just required some additional lemmas, which I had glossed over in doing the paper proof. I found that mechanizing the proofs typically increased my confidence in their correctness. Moreover, mechanized proofs are far more useful for keeping a clear and comprehensible record than informal proofs which are typically scattered across numerous sheets of paper or tex documents.”

Summary

- New, fully syntactic decision procedure for coproducts based on canonical forms methodology

Summary

- New, fully syntactic decision procedure for coproducts based on canonical forms methodology
- So far, straightforward to mechanize in Twelf

Summary

- New, fully syntactic decision procedure for coproducts based on canonical forms methodology
- So far, straightforward to mechanize in Twelf
- Accept Arbob to your PhD programs and he will do your proofs instead!

Thanks for listening!

The Twelf Wiki: `http://twelf.plparty.org`