RedPRL Documentation

Release

The RedPRL Development Team

Mar 22, 2018

Contents

   

Red**PRL** is an experimental proof assistant based on cubical computational type theory, which extends the Nuprl semantics by higher-dimensional features inspired by homotopy type theory. Red**PRL** is created and maintained by the RedPRL Development Team.

Red**PRL** is written in Standard ML, and is available for download on GitHub.

CHAPTER 1

Features

- computational canonicity and extraction
- univalence as a theorem
- strict (exact) equality types
- coequalizer and pushout types
- functional extensionality
- equality reflection
- proof tactics

CHAPTER 2

Papers & Talks

- Favonia. Cubical Computational Type Theory & RedPRL. 2018.
- Harper, Angiuli. Computational (Higher) Type Theory. ACM POPL Tutorial Session 2018.
- Angiuli, Harper, Wilson. Computational Higher-Dimensional Type Theory. POPL 2017.
- Sterling, Harper. Algebraic Foundations of Proof Refinement. Draft, 2016.

CHAPTER 3

RedPRL User Guide

3.1 Tutorial

We will walk through parts of `examples/tutorial.prl`, which was the live demo of Red**PRL** in our POPL 2018 tutorial on Computational (Higher) Type Theory. For further guidance, we recommend that new users consult the many other proofs in the `examples/` subdirectory.

Todo: Write the tutorial.

```
theorem Not :
  (-> bool bool)
by {
  lam b =>
  if b then `ff else `tt
}.

print Not.

// (not(not(b)) == b) because it holds for every closed boolean.
theorem NotNot :
  (->
   [b : bool]
   (= bool ($ Not ($ Not b)) b))
by {
  lam b =>
  // The next four lines can be replaced by auto.
  unfold Not;
  if b
  then (reduce at left; refine bool/eq/tt)
  else (reduce at left; refine bool/eq/ff)
}.

print NotNot.
```

```
// Type families respect equality proofs.
theorem RespectEquality :
  (->
   [family : (-> [b : bool] (U 0))]
   [b : bool]
   ($ family b)
   ($ family ($ Not ($ Not b))))
by {
  lam family b pf =>
  rewrite ($ NotNot b);
  [ with b' => `($ family b')
  , use pf
  ];
  auto
}.


// print does not mention the equality proof!
// (No need to ``transport'' at runtime.)
print RespectEquality.

// In fact, all proofs of (not(not(b)) == b) are equal.
theorem EqualityIrrelevant :
  (=
    (-> [b : bool] (= bool ($ Not ($ Not b)) b))
    NotNot
    (lam [b] ax))
by {
  auto
}.

print EqualityIrrelevant.

// Paths (cf equalities), like those arising from
// equivalences via univalence, do induce computation.
theorem FunToPair :
  (->
   [ty : (U 0 kan)]
   (-> bool ty)
   (* ty ty))
by {
  lam ty fun =>
  {`($ fun tt), `($ fun ff)}
}.

// {{{ Univalence

define HasAllPathsTo (#C,#c) = (-> [c' : #C] (path [_] #C c' #c)).
define IsContr (#C) = (* [c : #C] (HasAllPathsTo #C c)).
define Fiber (#A,#B,#f,#b) = (* [a : #A] (path [_] #B ($ #f a) #b)).
define IsEquiv (#A,#B,#f) = (-> [b : #B] (IsContr (Fiber #A #B #f b))).
define Equiv (#A,#B) = (* [f : (-> #A #B)] (IsEquiv #A #B f)).

theorem WeakConnection(#l:lvl) :
  (->
   [ty : (U #l hcom)]
   [a b : ty]
```

```
    [p : (path [_] ty a b)]
    (path [i] (path [_] ty (@ p i) b) p (abs [_] b)))
by {
   (lam ty a b p =>
     abs i j =>
       `(hcom 1~>0 ty b
         [i=0 [k] (hcom 0~>j ty (@ p k) [k=0 [w] (@ p w)] [k=1 [_] b])]
         [i=1 [k] (hcom 0~>1 ty (@ p k) [k=0 [w] (@ p w)] [k=1 [_] b])]
         [j=0 [k] (hcom 0~>i ty (@ p k) [k=0 [w] (@ p w)] [k=1 [_] b])]
         [j=1 [k] (hcom 0~>1 ty (@ p k) [k=0 [w] (@ p w)] [k=1 [_] b])]))
}.

tactic GetEndpoints(#p, #t:[exp,exp].tac) = {
  query pty <- #p;
  match pty  {
    [ty l r | #jdg{(path [_] %ty %l %r)} =>
      claim p/0 : (@ #p 0) = %l in %ty by {auto};
      claim p/1 : (@ #p 1) = %r in %ty by {auto};
      (#t p/0 p/1)
    ]
  }
}.


print WeakConnection.

theorem FunToPairIsEquiv :
   (->
    [ty : (U 0 kan)]
    (IsEquiv (-> bool ty) (* ty ty) ($ FunToPair ty)))
by {
   lam ty pair =>
   { { lam b => if b then `(!proj1 pair) else `(!proj2 pair)
     , abs _ => `pair }
   , unfold Fiber;
     lam {fun,p} =>
       (GetEndpoints p [p/0 p/1] #tac{
        (abs x =>
          {lam b => if b then `(!proj1 (@ p x)) else `(!proj2 (@ p x)),
           abs y =>
             `(@ ($ (WeakConnection #lvl{0}) (* ty ty) ($ FunToPair ty fun) pair p) x y)
          });
        [ unfold FunToPair in p/0; reduce in p/0 at right;
          inversion; with q3 q2 q1 q0 =>
            reduce at right in q2;
            reduce at right in q3;
            auto; with b =>
              elim b; reduce at right; symmetry; assumption
        , unfold FunToPair in p/1; reduce in p/1 at right;
          inversion; with q3 q2 q1 q0 => elim pair;
          reduce at right in q0; reduce at right in q1;
          auto; assumption
        ]
       })
   }
}.

theorem PathFunToPair :
```

```
  (->
   [ty : (U 0 kan)]
   (path [_] (U 0 kan) (-> bool ty) (* ty ty)))
by {
  lam ty => abs x =>
  `(V x (-> bool ty) (* ty ty)
    (tuple [proj1 ($ FunToPair ty)] [proj2 ($ FunToPairIsEquiv ty)]))
}.

// }}}

print PathFunToPair.

// We can coerce elements of (bool -> ty) to (ty * ty).
theorem RespectPaths :
  (->
   [ty : (U 0 kan)]
   (-> bool ty)
   (* ty ty))
by {
  lam ty fun =>
  `(coe 0~>1 [x] (@ ($ PathFunToPair ty) x) fun)
}.

print RespectPaths.

// When coercing, the choice of PathFunToPair matters!
theorem ComputeCoercion :
  (=
   (* bool bool)
   ($ RespectPaths bool (lam [b] b))
   (tuple [proj1 tt] [proj2 ff]))
by {
  auto
}.


// ---------------------------------------------------------
// Part Two
// ---------------------------------------------------------

// A constant path does not depend on its dimension.
theorem Refl :
  (->
   [ty : (U 0)]
   [a : ty]
   (path [_] ty a a))
by {
  lam ty a =>
  abs _ => `a
}.

// The path structure of each type is defined in terms of
// its constituent types.
theorem FunPath :
  (->
   [a b : (U 0)]
   [f g : (-> a b)]
```

```
    (path [_] (-> a b) f g)
    [arg : a]
    (path [_] b ($ f arg) ($ g arg)))
by {
  lam a b f g p =>
  lam arg => abs x =>
    `($ (@ p x) arg)
}.


print FunPath.

theorem PathInv :
  (->
   [ty : (U 0 kan)]
   [a b : ty]
   [p : (path [_] ty a b)]
   (path [_] ty b a))
by {
//        a           -- x
//     -------       |
//    |       |      y
//  p |       | a
//    |       |
//    b .... a

  lam ty a b p =>
  abs x =>
  `(hcom 0~>1 ty a [x=0 [y] (@ p y)] [x=1 [_] a])
}.

theorem PathConcat :
  (->
   [ty : (U 0 kan)]
   [a b c : ty]
   [p : (path [_] ty a b)]
   [q : (path [_] ty b c)]
   (path [_] ty a c))
by {
//        p           -- x
//     -------       |
//    |       |      y
//  a |       | q
//    |       |
//    a .... c

  lam ty a b c p q =>
  abs x =>
  `(hcom 0~>1 ty (@ p x) [x=0 [_] a] [x=1 [y] (@ q y)])
}.

theorem InvRefl :
  (->
   [ty : (U 0 kan)]
   [a : ty]
   (path
     [_] (path [_] ty a a)
     ($ PathInv ty a a (abs [_] a))
```

```
      (abs [_] a)))
by {
  // See diagram!
  lam ty a =>
  abs x y =>
  `(hcom 0~>1 ty a
    [x=0 [z] (hcom 0~>z ty a [y=0 [_] a] [y=1 [_] a])]
    [x=1 [_] a]
    [y=0 [_] a]
    [y=1 [_] a])
}.

// Although the path type is not defined by refl and J
// (as in HoTT), we can still define J using hcom + coe.
// The #l is an example of a parametrized definition.
theorem J(#l:lvl) :
  (->
   [ty : (U #l kan)]
   [a : ty]
   [fam : (-> [x : ty] (path [_] ty a x) (U #l kan))]
   [d : ($ fam a (abs [_] a))]
   [x : ty]
   [p : (path [_] ty a x)]
   ($ fam x p))
by {
  lam ty a fam d x p =>
  `(coe 0~>1
    [i] ($ fam
           (hcom 0~>1 ty a [i=0 [_] a] [i=1 [j] (@ p j)])
           (abs [j] (hcom 0~>j ty a [i=0 [_] a] [i=1 [j] (@ p j)]))) d)
}.

theorem JInv :
  (->
   [ty : (U 0 kan)]
   [a b : ty]
   [p : (path [_] ty a b)]
   (path [_] ty b a))
by {
  lam ty a b p =>
  exact
    ($ (J #lvl{0})
       ty
       a
       (lam [b _] (path [_] ty b a))
       (abs [_] a)
       b
       p)
  ; auto
  //; unfold J; reduce at left right; ?
}.

print JInv.

// Computing winding numbers in the circle.

// Bonus material:
```

```
theorem Shannon :
  (->
   [ty  : (-> bool (U 0))]
   [elt : (-> [b : bool] ($ ty b))]
   [b : bool]
   (= ($ ty b) ($ elt b) (if [b] ($ ty b) b ($ elt tt) ($ elt ff))))
by {
  lam ty elt b =>
  elim b; auto
}.
```

3.2 Language reference

Red**PRL** documents contain expressions written in multiple languages: the *top-level vernacular*, the *object language*, and the *tactic language*.

3.2.1 Top-level vernacular

The top-level vernacular is a very simple language of commands that interact with the *signature*: this language is for declaring *new theorems*, *definitional extensions* and *tactics*; the top-level vernacular can also be used to print out an object from the signature. This is the language that one writes in a `.prl` file.

Defining theorems

A *theorem* in Red**PRL** is given by a type (an object language expression) together with a tactic script which establishes that the given type is inhabited; when a theorem is declared, the tactic script is executed against the goal, and if the result is successful, the generated evidence is added to the signature.

```
theorem OpName(#p : ...) :
  // goal here (object language expression)
by {
  // script here (tactic expression)
}.
```

Most definitions in a Red**PRL** signature will take the form of theorems; but other forms of definition may be preferable, *depending on circumstances*.

Defining new operators

The most primitive way to define a new operator in Red**PRL** is to use the `define` command. A definition is specified by giving an operator name (which must be capitalized), together with a (possibly empty) sequence of parameters together with their valences, and an object-language term which shall be the definiens:

```
define OpName(#p : [dim].exp, ...) : exp =
  // object language expression here
.
```

A parameter is referenced using a *metavariable* (which is distinguished syntactically using the `#` sigil); the valence of a parameter specifies binding structure, with `[tau1,tau2].tau` being the valence of a binder of sort `tau` that binds a variable of sort `tau1` and a variable of sort `tau2`.

A simple definition of sort `exp` without parameters can be abbreviated as follows:

```
define OpName =
  // object language expression here
.
```

Definitions of this kind are not subject to any typing conditions in CHTT; instead, if you use a primitive definition within a proof, you will have to prove that it is well-typed.

Defining tactics

A tactic can be defined using the special `tactic` command:

```
tactic OpName(#p : ...) =
  // tactic expression here
.
```

This desugars to an instance of the `define` command, and differs only in that the body of the definiens is here parsed using the grammar of tactic expressions.

Printing objects

To print a previously-defined object from the signature, one can write the following command:

```
print OpName.
```

When to use theorems or definitions?

As a rule of thumb, in most cases it is simpler to interactively construct an element of a type using a `theorem` declaration than it is to define a code for an element, and then prove that it has the intended type. This is why theorems are usually preferred to definitions in Red**PRL**.

However, definitions may be preferable in some cases; consider the definition of an abbreviation for the type family `(lam [ty] (-> nat ty))` of sequences. As a theorem, this definition must take a universe level as a parameter

```
define Sequence(#l : lvl) :
  (-> [ty : (U #l)] (U #l))
by {
  // apply function introduction rule in the tactic language
  lam ty =>
    // explicitly give the body of the function in the object language
    `(-> nat ty)
}.
```

Later, when using this definition, one would have to explicitly provide the universe level, even though it does not play a part in the actual defined object: for instance, `(Sequence #lvl{0})`. The parameter was present only in order to express the type of the type family. On the other hand, with a definition, we can write the following:

```
define Sequence =
  (lam [ty] (-> nat ty))
.
```

One advantage of theorems over definitions is that Red**PRL** knows their type intrinsically; whereas definitions must be unfolded and proved to be well-typed at each use-site.

3.2.2 Object language

RedPRL's object language and tactic language share a common syntactic framework based on multi-sorted second-order abstract syntax, which provides a uniform treatment of binding with syntactic sorts. RedPRL has three main sorts: `exp` (the sort of expressions), `dim` (the sort of dimension expressions) and `tac` (the sort of tactic expressions).

The object language is written in a variant of s-expression notation, with binding operators written systematically in the style of `(lam [x] x)`. An expression in the object language is an *untyped program* or *realizer* in the language of Computational Higher Type Theory (CHTT).

These expressions include ordinary programming constructs like lambda abstraction and application, records, projection, etc., as well as cubical programming constructs inspired by cubical sets. Below are summarized common forms overlapping with other calculi.

Ordinary Operation	Expression
dependent function type	`(-> [x y ... : ty] ... ty)`
lambda abstraction	`(lam [x y ...] e)`
function application	`($ f e1 e2 ...)`
dependent record type	`(record [lbl ... : ty] ..)`
tuple (record element)	`(tuple [lbl e] ...)`
record projection	`(! lbl e)`

The cubical extension is characterized by a new sort of expressions, *dimension expressions* along with many new operations. A dimension expression can be a dimension variable `i`, representing an interval, or a dimension constant `0` or `1`, representing one of its end point.

Cubical Operation	Expression
coercion	`(coe r~>s [i] ty e)`
homogeneous composition	`(hcom r~>s ty cap [i=0 [j] tube0] ...)`
path type	`(path [i] ty e0 e1)`
line type	`(-> [i : dim] ... ty)`
path/line abstraction	`(abs [i j ...] e)`
path/line application	`(@ e r1 r2 ...)`
univalence	`(V a b e)`

Todo: Finish summary of object language terms.

3.2.3 Tactic language

Todo: Summarize tactic language

3.3 Atomic judgments

RedPRL currently has five forms of atomic (non-hypothetical) judgments that may appear in subgoals.

1. *Truth* asserts that a type is inhabited.
2. *Type equality* asserts an equality between two types.

3. *Subtyping* asserts a subtyping relation.
4. *Subkinding* asserts that some type is actually a universe in which all types has a particular kind.
5. *Term* lets the user give an expression.

Note that these judgment forms differ from our semantic presentations in papers.

3.3.1 Truth

A *truth* judgment

```
a true
```

or simply

```
a
```

means `a` is an inhabited type. Any inhabitant can realize this judgment. For example, the expression `1` realizes

```
int
```

because `1` is in the type `int`. This is commonly used to state a theorem or specify the type of the program to be implemented. In fact, all top-level theorems (see *Defining theorems*) must be in this judgmental form.

3.3.2 Type equality

A *type equality* judgment

```
a = b type
```

means `a` are `b` are equal types (without regard to universe level), and its realizer must be `ax`, the same as the realizer of equality types. For example, we have

```
int = int type
```

realized by `ax`. Multiverses are supported through kind markers such as `kan` or `discrete`:

```
a = b discrete type
a = b kan type
a = b coe type
a = b hcom type
a = b pre type
```

where `a = b kan type` means `a` and `b` are equal Kan types. (The judgment `a = b type` is really an abbreviation of `a = b pre type` because `pre` is the default kind.) Following the PRL family of proof assistants which use partial equivalence relations, well-typedness is defined as the equality of the type and itself; to save some typing, `a type` stands for `a = a type` and `a kan type` stands for `a = a kan type`.

In the presence of universes and equality types, one might wonder why we still have a dedicated judgmental form for type equality. That is, one may intuitively treat the judgment

```
a = b type
```

as `(= (U l) a b) true` for some unknown universe level `l`. It turns out to be very convenient to state type equality without specifying the universe levels; with this, we survived without a universe level synthesizer as the one in Nuprl, which was created to alleviate the burden of guessing universe levels.

3.3.3 Subtyping

A *subtype* judgment

```
a <= b type
```

states that `a` is a subtype of `b`. More precisely, the partial equivalence relation associated with `a` is a subrelation of the one associated with `b`. The realizer must be `ax`. There is no support of kind markers because the subtyping relation never takes additional structures into consideration.

This is currently used whenever we only need a subtyping relationship rather than type equality. For example, if a function `f` is in type `(-> a b)`, the rule to determine whether the function application `($ f x)` is in type `b'` will only demand `b <= b' type` rather than `b = b' type`. That said, the only non-trivial subtyping relation one can prove in Red**PRL** now is the cumulativity of universes. One instance would be

```
(U 0 discrete) <= (U 1 kan)
```

realized by `ax`.

3.3.4 Subkinding

The following are *subkind* judgments:

```
a <= discrete universe
a <= kan universe
a <= coe universe
a <= hcom universe
a <= pre universe
```

They assert that `a` is a subuniverse of the universe of the specified kind at the omega level. Intuitively, `a <= k universe` would be the *subtyping judgment* `a <= (U omega k) type` if we could internalize universes at the omega level. The realizer must be `ax`. These judgments play the same role as *subtyping judgments* except that they handle the cases where the right hand side is some omega-level universe. Suppose a function `f` is in type `(-> a b)`. The rule to determine whether the function application `($ f x)` is a type will demand `b <= pre universe` rather than `b = (U omega) type` (or `b = (U l) type` for some universe level `l`).

3.3.5 Term

A *term* judgment is displayed in the sort of the expression it is asking for, for example:

```
dim
exp
```

The realizer is the received term from the user. This is used to obtain motives or dimension expressions. For example, the `rewrite` tactic requires users to specify the parts to be rewritten by fulfilling *term* subgoals.

3.4 Multiverses

Todo: To Infinity. . . and Beyond!

3.5 Refinement rules

Todo: Fill in the refinement rules listed below.

3.5.1 Booleans

bool/eqtype

```
H >> bool = bool in (U #l #k)
```

bool/eq/tt

```
H >> tt = tt in bool
```

bool/eq/ff

```
H >> ff = ff in bool
```

bool/eq/if

```
H >> (if [x] (#c0 x) #m0 #t0 #f0) = (if [x] (#c1 x) #m1 #t1 #f1) in #ty
where H >> #m0 = #m1 synth ~> bool, psi
| H >> #t0 = #t1 in (#c0 tt)
| H >> #f0 = #f1 in (#c0 ff)
| H, x:bool >> #c0 = #c1 type
| psi
| H >> (#c0 #m0) <= #ty type
```

3.5.2 Natural numbers and integers

nat/eqtype

nat/eq/zero

nat/eq/succ

nat/eq/nat-rec

int/eqtype

int/eq/pos

int/eq/negsucc

int/eq/int-rec

3.5.3 Void

void/eqtype

3.5.4 Circle

s1/eqtype

s1/eq/base

s1/eq/loop

s1/eq/fcom

s1/eq/s1-rec

s1/beta/loop

3.5.5 Dependent functions

fun/eqtype

```
H >> (-> [x : #a0] (#b0 x)) = (-> [x : #a1] (#b1 x)) in (U #l #k)
where
  (#k/dom, #k/cod) <-
    (discrete, discrete) if #k == discrete
    (coe, kan) if #k == kan
    (pre, hcom) if #k == hcom
    (coe, coe) if #k == coe
    (pre, pre) if #k == pre
| H >> #a0 = #a1 in (U #l #k/dom)
| H, x:#a0 >> (#b0 x) = (#b1 x) in (U #l #k/cod)
```

fun/eq/lam

```
H >> (lam [x] (#e0 x)) = (lam [x] (#e1 x)) in (-> [x : #a] (#b x))
| H, x:#a >> (#e0 x) = (#e1 x) in (#b x)
| H >> #a type
```

fun/intro

```
H >> (-> [x : #a] (#b x)) ext (lam [x] (#e x))
| H, x:#a >> (#b x) ext (#e x)
| H >> #a type
```

fun/eq/eta

```
H >> #e = #f in (-> [x : #a] (#b x))
| H >> (lam [x] ($ #e x)) = #f in (-> [x : #a] (#b x))
| H >> #e = #e in (-> [x : #a] (#b x))
```

fun/eq/app

```
H >> ($ #f0 #e0) = ($ #f1 #e1) in #ty
where H >> #f0 = #f1 synth ~> (-> [x : #a] (#b x)), psi
| H >> #e0 = #e1 in #a
| psi
| H >> (#cod #e0) <= #ty type
```

3.5.6 Records

record/eqtype

```
H >> (record [lbl/a : #a0] ... [lbl/b : (#b0 lbl/a ...)])
     = (record [lbl/a : #a1] ... [lbl/b : (#b1 lbl/a ...)])
     in (U #l #k)
where
  (#k/hd, #kltl) <-
    (discrete, discrete) if #k == discrete
    (kan, kan) if #k == kan
    (hcom, kan) if #k == hcom
    (coe, coe) if #k == coe
    (pre, pre) if #k == pre
| H >> #a0 = #a1 in (U #l #k/hd)
| ...
| H, x : #a0, ... >> (#b0 x ...) = (#b1 x ...) in (U #l #k/tl)
```

Todo: The choice of kinds `#k/hd` and `#k/tl` looks a little fishy; is this exactly what would be generated if a record were encoded as an iterated sigma type?

record/eq/tuple

```
H >> (tuple [lbl/a #p0] ... [lbl/b #q0])
     = (tuple [lbl/a #p1] ... [lbl/b #q1])
     in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> #p0 = #p1 in #a
| ...
| H >> #q0 = #q1 in (#b #p0 ...)
| ...
| H, x:#a, ... >> (#b x ...) type
```

record/eq/eta

```
H >> #e0 = #e1 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> (tuple [lbl/a (! lbl/a #e0)] ... [lbl/b (! lbl/b #e0)])
|       = #e1 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
| H >> #e0 in (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
```

record/eq/proj

```
H >> (! lbl #e0) = (! lbl #e1) in #ty
where H >> #e0 = #e1 synth ~> (record [lbl0 : #a0] ... [lbl : (#a ...)] ...), psi
| psi
| H >> (#a (! lbl0  #e0) ...) <= #ty type
```

record/intro

```
H >> (record [lbl/a : #a] ... [lbl/b : (#b lbl/a ...)])
     ext (tuple [lbl/a #p/a] ... [lbl/b #p/b])
| H >> #a ext #p/a
| ...
| H >> (#b #p/a ...) ext #p/b
| ...
| H, x:#a, ... >> (#b x ...) type
```

3.5.7 Paths

path/eqtype

path/eq/abs

path/intro

path/eq/eta

path/eq/app

path/eq/app/const

path/eq/from-line

3.5.8 Lines

line/eqtype

line/eq/abs

line/intro

line/eq/eta

line/eq/app

3.5.9 Pushouts

pushout/eqtype

pushout/eq/left

pushout/eq/right

pushout/eq/glue

pushout/eq/fcom

pushout/eq/pushout-rec

pushout/beta/glue

3.5.10 Coequalizers

coeq/eqtype

coeq/eq/cod

coeq/eq/dom

coeq/eq/fcom

coeq/beta/dom

coeq/eq/coeq-rec

- search

3.7 Acknowledgments

This research was sponsored by the Air Force Office of Scientific Research under grant number FA9550-15-1-0053 and the National Science Foundation under grant number DMS-1638352. We also thank the Isaac Newton Institute for Mathematical Sciences for its support and hospitality during the program "Big Proof" when part of this work was undertaken; the program was supported by the Engineering and Physical Sciences Research Council under grant number EP/K032208/1. The views and conclusions contained here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, government or any other entity.

Index

S

U

V