```
// ::::::::::::::
// abstract.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <utility>

#include "kernel/expr.h"

namespace lean {
/** \brief Replace the free variables s[0], ..., s[n-1] in e with bound
 * variables bvar(n-1), ..., bvar(0). */
expr abstract(expr const &e, unsigned n, expr const *s);
inline expr abstract(expr const &e, expr const &s) {
    return abstract(e, 1, &s);
}
expr abstract(expr const &e, name const &n);

}  // namespace lean
// ::::::::::::::
// cache_stack.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <lean/debug.h>

#include <memory>
#include <vector>

/** \brief Macro for creating a stack of objects of type Cache in thread local
   storage. The argument \c Arg is provided to every new instance of Cache. The
   macro provides the helper class Cache_ref that "reuses" cache objects from
   the stack.
*/
#define MK_CACHE_STACK(Cache, Arg)                               \
    struct Cache##_stack {                                       \
        unsigned m_top;                                          \
        std::vector<std::unique_ptr<Cache>> m_cache_stack;       \
        Cache##_stack() : m_top(0) {}                            \
    };                                                           \
    MK_THREAD_LOCAL_GET_DEF(Cache##_stack, get_##Cache##_stack); \
    class Cache##_ref {                                          \
        Cache *m_cache;                                          \
                                                                 \
       public:                                                   \
        Cache##_ref() {                                          \
            Cache##_stack &s = get_##Cache##_stack();            \
            lean_assert(s.m_top <= s.m_cache_stack.size());      \
            if (s.m_top == s.m_cache_stack.size())               \
                s.m_cache_stack.push_back(                       \
                    std::unique_ptr<Cache>(new Cache(Arg)));     \
            m_cache = s.m_cache_stack[s.m_top].get();            \
            s.m_top++;                                           \
        }                                                        \
        ~Cache##_ref() {                                         \
            Cache##_stack &s = get_##Cache##_stack();            \
            lean_assert(s.m_top > 0);                            \
            s.m_top--;                                           \
            m_cache->clear();                                    \
```

```
        }                                                   \
        Cache *operator->() const { return m_cache; }       \
    };
// ::::::::::::::
// declaration.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <algorithm>
#include <limits>
#include <string>

#include "kernel/expr.h"

namespace lean {
/**
inductive reducibility_hints
| opaque  : reducibility_hints
| abbrev  : reducibility_hints
| regular : nat → reducibility_hints

Reducibility hints are used in the convertibility checker (aka is_def_eq
predicate), whenever checking a constraint such as

           (f ...) =?= (g ...)

where f and g are definitions, and the checker has to decide which one will be
unfolded. If f (g) is Opaque,             then g (f) is unfolded if it is also
not marked as Opaque. Else if f (g) is Abbreviation,  then f (g) is unfolded if
g (f) is also not marked as Abbreviation. Else if f and g are Regular,    then
we unfold the one with the biggest definitional height. Otherwise unfold both.

The definitional height is by default computed by the kernel. It only takes into
account other Regular definitions used in a definition.

Remark: the hint only affects performance. */
enum class reducibility_hints_kind { Opaque, Abbreviation, Regular };
class reducibility_hints : public object_ref {
    reducibility_hints(b_obj_arg o, bool b) : object_ref(o, b) {}
    explicit reducibility_hints(object *r) : object_ref(r) {}

   public:
    static reducibility_hints mk_opaque() {
        return reducibility_hints(
            box(static_cast<unsigned>(reducibility_hints_kind::Opaque)));
    }
    static reducibility_hints mk_abbreviation() {
        return reducibility_hints(
            box(static_cast<unsigned>(reducibility_hints_kind::Abbreviation)));
    }
    static reducibility_hints mk_regular(unsigned h);
    reducibility_hints_kind kind() const {
        return static_cast<reducibility_hints_kind>(obj_tag(raw()));
    }
    bool is_regular() const {
        return kind() == reducibility_hints_kind::Regular;
    }
    unsigned get_height() const;
};

/** Given h1 and h2 the hints for definitions f1 and f2, then
    result is
    <  0 If f1 should be unfolded
    == 0 If f1 and f2 should be unfolded
    >  0 If f2 should be unfolded */
```

```
int compare(reducibility_hints const &h1, reducibility_hints const &h2);

/*
structure constant_val :=
(id : name) (lparams : list name) (type : expr)
*/
class constant_val : public object_ref {
   public:
    constant_val(name const &n, names const &lparams, expr const &type);
    constant_val(constant_val const &other) : object_ref(other) {}
    constant_val(constant_val &&other) : object_ref(other) {}
    constant_val &operator=(constant_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val &operator=(constant_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    name const &get_name() const {
        return static_cast<name const &>(cnstr_get_ref(*this, 0));
    }
    names const &get_lparams() const {
        return static_cast<names const &>(cnstr_get_ref(*this, 1));
    }
    expr const &get_type() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 2));
    }
};

/*
structure axiom_val extends constant_val :=
(is_unsafe : bool)
*/
class axiom_val : public object_ref {
   public:
    axiom_val(name const &n, names const &lparams, expr const &type,
              bool is_unsafe);
    axiom_val(axiom_val const &other) : object_ref(other) {}
    axiom_val(axiom_val &&other) : object_ref(other) {}
    axiom_val &operator=(axiom_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    axiom_val &operator=(axiom_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    bool is_unsafe() const;
};

enum class definition_safety { unsafe, safe, partial };

/*
structure definition_val extends constant_val :=
(value : expr) (hints : reducibility_hints) (is_unsafe : bool)
*/
class definition_val : public object_ref {
   public:
    definition_val(name const &n, names const &lparams, expr const &type,
                   expr const &val, reducibility_hints const &hints,
                   definition_safety safety);
    definition_val(definition_val const &other) : object_ref(other) {}
    definition_val(definition_val &&other) : object_ref(other) {}
```

```cpp
    definition_val &operator=(definition_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    definition_val &operator=(definition_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    expr const &get_value() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 1));
    }
    reducibility_hints const &get_hints() const {
        return static_cast<reducibility_hints const &>(cnstr_get_ref(*this, 2));
    }
    definition_safety get_safety() const;
    bool is_unsafe() const { return get_safety() == definition_safety::unsafe; }
};
typedef list_ref<definition_val> definition_vals;

/*
structure theorem_val extends constant_val :=
(value : task expr)
*/
class theorem_val : public object_ref {
   public:
    theorem_val(name const &n, names const &lparams, expr const &type,
                expr const &val);
    theorem_val(theorem_val const &other) : object_ref(other) {}
    theorem_val(theorem_val &&other) : object_ref(other) {}
    theorem_val &operator=(theorem_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    theorem_val &operator=(theorem_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    expr const &get_value() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 1));
    }
};

/*
structure opaque_val extends constant_val :=
(value : expr)
*/
class opaque_val : public object_ref {
   public:
    opaque_val(name const &n, names const &lparams, expr const &type,
               expr const &val, bool is_unsafe);
    opaque_val(opaque_val const &other) : object_ref(other) {}
    opaque_val(opaque_val &&other) : object_ref(other) {}
    opaque_val &operator=(opaque_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    opaque_val &operator=(opaque_val &&other) {
        object_ref::operator=(other);
```

```
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    expr const &get_value() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 1));
    }
    bool is_unsafe() const;
};

/*
structure constructor :=
(id : name) (type : expr)
*/
typedef pair_ref<name, expr> constructor;
inline name const &constructor_name(constructor const &c) { return c.fst(); }
inline expr const &constructor_type(constructor const &c) { return c.snd(); }
typedef list_ref<constructor> constructors;

/**
structure inductive_type where
(id : name) (type : expr) (cnstrs : list constructor)
*/
class inductive_type : public object_ref {
   public:
    inductive_type(name const &id, expr const &type,
                   constructors const &cnstrs);
    inductive_type(inductive_type const &other) : object_ref(other) {}
    inductive_type(inductive_type &&other) : object_ref(other) {}
    inductive_type &operator=(inductive_type const &other) {
        object_ref::operator=(other);
        return *this;
    }
    inductive_type &operator=(inductive_type &&other) {
        object_ref::operator=(other);
        return *this;
    }
    name const &get_name() const {
        return static_cast<name const &>(cnstr_get_ref(*this, 0));
    }
    expr const &get_type() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 1));
    }
    constructors const &get_cnstrs() const {
        return static_cast<constructors const &>(cnstr_get_ref(*this, 2));
    }
};
typedef list_ref<inductive_type> inductive_types;

/*
inductive declaration
| axiom_decl       (val : axiom_val)
| defn_decl        (val : definition_val)
| thm_decl         (val : theorem_val)
| opaque_decl      (val : opaque_val)
| quot_decl        (id : name)
| mutual_defn_decl (defns : list definition_val) -- All definitions must be
marked as `unsafe` | induct_decl      (lparams : list name) (nparams : nat)
(types : list inductive_type) (is_unsafe : bool)
*/
enum class declaration_kind {
    Axiom,
    Definition,
    Theorem,
    Opaque,
    Quot,
```

```
    MutualDefinition,
    Inductive
};
class declaration : public object_ref {
    object *get_val_obj() const { return cnstr_get(raw(), 0); }
    object_ref const &to_val() const { return cnstr_get_ref(*this, 0); }

   public:
    declaration();
    declaration(declaration const &other) : object_ref(other) {}
    declaration(declaration &&other) : object_ref(other) {}
    /* low-level constructors */
    explicit declaration(object *o) : object_ref(o) {}
    explicit declaration(b_obj_arg o, bool b) : object_ref(o, b) {}
    explicit declaration(object_ref const &o) : object_ref(o) {}
    declaration_kind kind() const {
        return static_cast<declaration_kind>(obj_tag(raw()));
    }

    declaration &operator=(declaration const &other) {
        object_ref::operator=(other);
        return *this;
    }
    declaration &operator=(declaration &&other) {
        object_ref::operator=(other);
        return *this;
    }

    friend bool is_eqp(declaration const &d1, declaration const &d2) {
        return d1.raw() == d2.raw();
    }

    bool is_definition() const {
        return kind() == declaration_kind::Definition;
    }
    bool is_axiom() const { return kind() == declaration_kind::Axiom; }
    bool is_theorem() const { return kind() == declaration_kind::Theorem; }
    bool is_opaque() const { return kind() == declaration_kind::Opaque; }
    bool is_mutual() const {
        return kind() == declaration_kind::MutualDefinition;
    }
    bool is_inductive() const { return kind() == declaration_kind::Inductive; }
    bool is_unsafe() const;
    bool has_value() const { return is_theorem() || is_definition(); }

    axiom_val const &to_axiom_val() const {
        lean_assert(is_axiom());
        return static_cast<axiom_val const &>(cnstr_get_ref(raw(), 0));
    }
    definition_val const &to_definition_val() const {
        lean_assert(is_definition());
        return static_cast<definition_val const &>(cnstr_get_ref(raw(), 0));
    }
    theorem_val const &to_theorem_val() const {
        lean_assert(is_theorem());
        return static_cast<theorem_val const &>(cnstr_get_ref(raw(), 0));
    }
    opaque_val const &to_opaque_val() const {
        lean_assert(is_opaque());
        return static_cast<opaque_val const &>(cnstr_get_ref(raw(), 0));
    }
    definition_vals const &to_definition_vals() const {
        lean_assert(is_mutual());
        return static_cast<definition_vals const &>(cnstr_get_ref(raw(), 0));
    }
};

inline optional<declaration> none_declaration() {
    return optional<declaration>();
}
```

```
inline optional<declaration> some_declaration(declaration const &o) {
    return optional<declaration>(o);
}
inline optional<declaration> some_declaration(declaration &&o) {
    return optional<declaration>(std::forward<declaration>(o));
}

declaration mk_definition(name const &n, names const &lparams, expr const &t,
                          expr const &v, reducibility_hints const &hints,
                          definition_safety safety = definition_safety::safe);
declaration mk_definition(environment const &env, name const &n,
                          names const &lparams, expr const &t, expr const &v,
                          definition_safety safety = definition_safety::safe);
declaration mk_opaque(name const &n, names const &lparams, expr const &t,
                      expr const &v, bool unsafe);
declaration mk_axiom(name const &n, names const &lparams, expr const &t,
                     bool unsafe = false);
declaration mk_inductive_decl(names const &lparams, nat const &nparams,
                              inductive_types const &types, bool is_unsafe);

/** \brief Similar to mk_definition but infer the value of unsafe flag.
    That is, set it to true if \c t or \c v contains a unsafe declaration. */
declaration mk_definition_inferring_unsafe(environment const &env,
                                           name const &n, names const &lparams,
                                           expr const &t, expr const &v,
                                           reducibility_hints const &hints);
declaration mk_definition_inferring_unsafe(environment const &env,
                                           name const &n, names const &lparams,
                                           expr const &t, expr const &v);
/** \brief Similar to mk_axiom but infer the value of unsafe flag.
    That is, set it to true if \c t or \c v contains a unsafe declaration. */
declaration mk_axiom_inferring_unsafe(environment const &env, name const &n,
                                      names const &lparams, expr const &t);

/** \brief View for manipulating declaration.induct_decl constructor.
    | induct_decl      (lparams : list name) (nparams : nat) (types : list
   inductive_type) (is_unsafe : bool) */
class inductive_decl : public object_ref {
   public:
    inductive_decl(inductive_decl const &other) : object_ref(other) {}
    inductive_decl(inductive_decl &&other) : object_ref(other) {}
    inductive_decl(declaration const &d) : object_ref(d) {
        lean_assert(d.is_inductive());
    }
    inductive_decl &operator=(inductive_decl const &other) {
        object_ref::operator=(other);
        return *this;
    }
    inductive_decl &operator=(inductive_decl &&other) {
        object_ref::operator=(other);
        return *this;
    }
    names const &get_lparams() const {
        return static_cast<names const &>(cnstr_get_ref(raw(), 0));
    }
    nat const &get_nparams() const {
        return static_cast<nat const &>(cnstr_get_ref(raw(), 1));
    }
    inductive_types const &get_types() const {
        return static_cast<inductive_types const &>(cnstr_get_ref(raw(), 2));
    }
    bool is_unsafe() const;
};

/*
structure inductive_val extends constant_val where
(nparams : nat)       -- Number of parameters
(nindices : nat)      -- Number of indices
(all : list name)     -- List of all (including this one) inductive datatypes in
the mutual declaration containing this one (cnstrs : list name)  -- List of all
```

```
constructors for this inductive datatype (is_rec : bool)       -- `tt` iff it is
recursive (is_unsafe : bool) (is_reflexive : bool)
*/
class inductive_val : public object_ref {
   public:
    inductive_val(name const &n, names const &lparams, expr const &type,
                  unsigned nparams, unsigned nindices, names const &all,
                  names const &cnstrs, bool is_rec, bool is_unsafe,
                  bool is_reflexive, bool is_nested);
    inductive_val(inductive_val const &other) : object_ref(other) {}
    inductive_val(inductive_val &&other) : object_ref(other) {}
    inductive_val &operator=(inductive_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    inductive_val &operator=(inductive_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    unsigned get_nparams() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 1))
            .get_small_value();
    }
    unsigned get_nindices() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 2))
            .get_small_value();
    }
    names const &get_all() const {
        return static_cast<names const &>(cnstr_get_ref(*this, 3));
    }
    names const &get_cnstrs() const {
        return static_cast<names const &>(cnstr_get_ref(*this, 4));
    }
    unsigned get_ncnstrs() const { return length(get_cnstrs()); }
    bool is_rec() const;
    bool is_unsafe() const;
    bool is_reflexive() const;
    bool is_nested() const;
};

/*
structure constructor_val extends constant_val :=
(induct  : name)  -- Inductive type this constructor is a member of
(cidx    : nat)   -- Constructor index (i.e., position in the inductive
declaration) (nparams : nat)   -- Number of parameters in inductive datatype
`induct` (nfields : nat)   -- Number of fields (i.e., arity - nparams)
(is_unsafe : bool)
*/
class constructor_val : public object_ref {
   public:
    constructor_val(name const &n, names const &lparams, expr const &type,
                    name const &induct, unsigned cidx, unsigned nparams,
                    unsigned nfields, bool is_unsafe);
    constructor_val(constructor_val const &other) : object_ref(other) {}
    constructor_val(constructor_val &&other) : object_ref(other) {}
    constructor_val &operator=(constructor_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    constructor_val &operator=(constructor_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_induct() const {
```

```
        return static_cast<name const &>(cnstr_get_ref(*this, 1));
    }
    unsigned get_cidx() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 2))
            .get_small_value();
    }
    unsigned get_nparams() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 3))
            .get_small_value();
    }
    unsigned get_nfields() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 4))
            .get_small_value();
    }
    bool is_unsafe() const;
};

/*
structure recursor_rule :=
(cnstr : name)  -- Reduction rule for this constructor
(nfields : nat) -- Number of fields (i.e., without counting inductive datatype
parameters) (rhs : expr)    -- Right hand side of the reduction rule
*/
class recursor_rule : public object_ref {
   public:
    recursor_rule(name const &cnstr, unsigned nfields, expr const &rhs);
    recursor_rule(recursor_rule const &other) : object_ref(other) {}
    recursor_rule(recursor_rule &&other) : object_ref(other) {}
    recursor_rule &operator=(recursor_rule const &other) {
        object_ref::operator=(other);
        return *this;
    }
    recursor_rule &operator=(recursor_rule &&other) {
        object_ref::operator=(other);
        return *this;
    }
    name const &get_cnstr() const {
        return static_cast<name const &>(cnstr_get_ref(*this, 0));
    }
    unsigned get_nfields() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 1))
            .get_small_value();
    }
    expr const &get_rhs() const {
        return static_cast<expr const &>(cnstr_get_ref(*this, 2));
    }
};

typedef list_ref<recursor_rule> recursor_rules;

/*
structure recursor_val extends constant_val :=
(all : list name)             -- List of all inductive datatypes in the mutual
declaration that generated this recursor (nparams : nat)              -- Number
of parameters (nindices : nat)             -- Number of indices (nmotives : nat)
-- Number of motives (nminors : nat)              -- Number of minor premises
(rules : list recursor_rule) -- A reduction for each constructor
(k : bool)                   -- It supports K-like reduction
(is_unsafe : bool)
*/
class recursor_val : public object_ref {
   public:
    recursor_val(name const &n, names const &lparams, expr const &type,
                 names const &all, unsigned nparams, unsigned nindices,
                 unsigned nmotives, unsigned nminors,
                 recursor_rules const &rules, bool k, bool is_unsafe);
    recursor_val(recursor_val const &other) : object_ref(other) {}
    recursor_val(recursor_val &&other) : object_ref(other) {}
    recursor_val &operator=(recursor_val const &other) {
        object_ref::operator=(other);
```

```
        return *this;
    }
    recursor_val &operator=(recursor_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
    name const &get_induct() const { return get_name().get_prefix(); }
    names const &get_all() const {
        return static_cast<names const &>(cnstr_get_ref(*this, 1));
    }
    unsigned get_nparams() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 2))
            .get_small_value();
    }
    unsigned get_nindices() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 3))
            .get_small_value();
    }
    unsigned get_nmotives() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 4))
            .get_small_value();
    }
    unsigned get_nminors() const {
        return static_cast<nat const &>(cnstr_get_ref(*this, 5))
            .get_small_value();
    }
    unsigned get_major_idx() const {
        return get_nparams() + get_nmotives() + get_nminors() + get_nindices();
    }
    recursor_rules const &get_rules() const {
        return static_cast<recursor_rules const &>(cnstr_get_ref(*this, 6));
    }
    bool is_k() const;
    bool is_unsafe() const;
};

enum class quot_kind { Type, Mk, Lift, Ind };

/*
inductive quot_kind
| type  -- `quot`
| cnstr -- `quot.mk`
| lift  -- `quot.lift`
| ind   -- `quot.ind`

structure quot_val extends constant_val :=
(kind : quot_kind)
*/
class quot_val : public object_ref {
   public:
    quot_val(name const &n, names const &lparams, expr const &type,
             quot_kind k);
    quot_val(quot_val const &other) : object_ref(other) {}
    quot_val(quot_val &&other) : object_ref(other) {}
    quot_val &operator=(quot_val const &other) {
        object_ref::operator=(other);
        return *this;
    }
    quot_val &operator=(quot_val &&other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
    }
    name const &get_name() const { return to_constant_val().get_name(); }
```

```
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    quot_kind get_quot_kind() const;
};

/*
/-- Information associated with constant declarations. -/
inductive constant_info
| axiom_info    (val : axiom_val)
| defn_info     (val : definition_val)
| thm_info      (val : theorem_val)
| opaque_info   (val : opaque_val)
| quot_info     (val : quot_val)
| induct_info   (val : inductive_val)
| cnstr_info    (val : constructor_val)
| rec_info      (val : recursor_val)
*/
enum class constant_info_kind {
    Axiom,
    Definition,
    Theorem,
    Opaque,
    Quot,
    Inductive,
    Constructor,
    Recursor
};
class constant_info : public object_ref {
    object *get_val_obj() const { return cnstr_get(raw(), 0); }
    object_ref const &to_val() const { return cnstr_get_ref(*this, 0); }
    constant_val const &to_constant_val() const {
        return static_cast<constant_val const &>(cnstr_get_ref(to_val(), 0));
    }

   public:
    constant_info();
    constant_info(declaration const &d);
    constant_info(definition_val const &v);
    constant_info(quot_val const &v);
    constant_info(inductive_val const &v);
    constant_info(constructor_val const &v);
    constant_info(recursor_val const &v);
    constant_info(constant_info const &other) : object_ref(other) {}
    constant_info(constant_info &&other) : object_ref(other) {}
    explicit constant_info(b_obj_arg o, bool b) : object_ref(o, b) {}
    explicit constant_info(obj_arg o) : object_ref(o) {}

    constant_info_kind kind() const {
        return static_cast<constant_info_kind>(cnstr_tag(raw()));
    }

    constant_info &operator=(constant_info const &other) {
        object_ref::operator=(other);
        return *this;
    }
    constant_info &operator=(constant_info &&other) {
        object_ref::operator=(other);
        return *this;
    }

    friend bool is_eqp(constant_info const &d1, constant_info const &d2) {
        return d1.raw() == d2.raw();
    }

    bool is_unsafe() const;

    bool is_definition() const {
        return kind() == constant_info_kind::Definition;
    }
    bool is_axiom() const { return kind() == constant_info_kind::Axiom; }
```

```
    bool is_theorem() const { return kind() == constant_info_kind::Theorem; }
    bool is_opaque() const { return kind() == constant_info_kind::Opaque; }
    bool is_inductive() const {
        return kind() == constant_info_kind::Inductive;
    }
    bool is_constructor() const {
        return kind() == constant_info_kind::Constructor;
    }
    bool is_recursor() const { return kind() == constant_info_kind::Recursor; }
    bool is_quot() const { return kind() == constant_info_kind::Quot; }

    name const &get_name() const { return to_constant_val().get_name(); }
    names const &get_lparams() const { return to_constant_val().get_lparams(); }
    unsigned get_num_lparams() const { return length(get_lparams()); }
    expr const &get_type() const { return to_constant_val().get_type(); }
    bool has_value(bool allow_opaque = false) const {
        return is_theorem() || is_definition() || (allow_opaque && is_opaque());
    }
    reducibility_hints const &get_hints() const;

    axiom_val const &to_axiom_val() const {
        lean_assert(is_axiom());
        return static_cast<axiom_val const &>(to_val());
    }
    definition_val const &to_definition_val() const {
        lean_assert(is_definition());
        return static_cast<definition_val const &>(to_val());
    }
    theorem_val const &to_theorem_val() const {
        lean_assert(is_theorem());
        return static_cast<theorem_val const &>(to_val());
    }
    opaque_val const &to_opaque_val() const {
        lean_assert(is_opaque());
        return static_cast<opaque_val const &>(to_val());
    }
    inductive_val const &to_inductive_val() const {
        lean_assert(is_inductive());
        return static_cast<inductive_val const &>(to_val());
    }
    constructor_val const &to_constructor_val() const {
        lean_assert(is_constructor());
        return static_cast<constructor_val const &>(to_val());
    }
    recursor_val const &to_recursor_val() const {
        lean_assert(is_recursor());
        return static_cast<recursor_val const &>(to_val());
    }
    quot_val const &to_quot_val() const {
        lean_assert(is_quot());
        return static_cast<quot_val const &>(to_val());
    }

    expr get_value(bool DEBUG_CODE(allow_opaque)) const {
        lean_assert(has_value(allow_opaque));
        if (is_theorem())
            return to_theorem_val().get_value();
        else
            return static_cast<expr const &>(cnstr_get_ref(to_val(), 1));
    }
    expr get_value() const { return get_value(false); }
};

inline optional<constant_info> none_constant_info() {
    return optional<constant_info>();
}
inline optional<constant_info> some_constant_info(constant_info const &o) {
    return optional<constant_info>(o);
}
inline optional<constant_info> some_constant_info(constant_info &&o) {
```

```
    return optional<constant_info>(std::forward<constant_info>(o));
}

static_assert(static_cast<unsigned>(declaration_kind::Axiom) ==
                  static_cast<unsigned>(constant_info_kind::Axiom),
              "declaration vs constant_info tag mismatch");
static_assert(static_cast<unsigned>(declaration_kind::Definition) ==
                  static_cast<unsigned>(constant_info_kind::Definition),
              "declaration vs constant_info tag mismatch");
static_assert(static_cast<unsigned>(declaration_kind::Theorem) ==
                  static_cast<unsigned>(constant_info_kind::Theorem),
              "declaration vs constant_info tag mismatch");

void initialize_declaration();
void finalize_declaration();
}  // namespace lean
// ::::::::::::::
// environment.h
// ::::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <lean/optional.h>

#include <memory>
#include <utility>
#include <vector>

#include "kernel/declaration.h"
#include "kernel/expr.h"
#include "util/list.h"
#include "util/name_map.h"
#include "util/name_set.h"
#include "util/rb_map.h"
#include "util/rc.h"

#ifndef LEAN_BELIEVER_TRUST_LEVEL
/* If an environment E is created with a trust level >
   LEAN_BELIEVER_TRUST_LEVEL, then we can add declarations to E without type
   checking them. */
#define LEAN_BELIEVER_TRUST_LEVEL 1024
#endif

namespace lean {
class environment_extension {
   public:
    virtual ~environment_extension() {}
};

class environment : public object_ref {
    friend class add_inductive_fn;

    void check_name(name const &n) const;
    void check_duplicated_univ_params(names ls) const;

    void add_core(constant_info const &info);
    void mark_quot_initialized();
    environment add(constant_info const &info) const;
    environment add_axiom(declaration const &d, bool check) const;
    environment add_definition(declaration const &d, bool check) const;
    environment add_theorem(declaration const &d, bool check) const;
    environment add_opaque(declaration const &d, bool check) const;
    environment add_mutual(declaration const &d, bool check) const;
    environment add_quot() const;
    environment add_inductive(declaration const &d) const;

```

```
   public:
    environment(unsigned trust_lvl = 0);
    environment(environment const &other) : object_ref(other) {}
    environment(environment &&other) : object_ref(other) {}
    explicit environment(b_obj_arg o, bool b) : object_ref(o, b) {}
    explicit environment(obj_arg o) : object_ref(o) {}
    ~environment() {}

    environment &operator=(environment const &other) {
        object_ref::operator=(other);
        return *this;
    }
    environment &operator=(environment &&other) {
        object_ref::operator=(other);
        return *this;
    }

    /** \brief Return the trust level of this environment. */
    unsigned trust_lvl() const;

    bool is_quot_initialized() const;

    void set_main_module(name const &n);

    name get_main_module() const;

    /** \brief Return information for the constant with name \c n (if it is
     * defined in this environment). */
    optional<constant_info> find(name const &n) const;

    /** \brief Return information for the constant with name \c n. Throws and
     * exception if constant declaration does not exist in this environment. */
    constant_info get(name const &n) const;

    /** \brief Extends the current environment with the given declaration */
    environment add(declaration const &d, bool check = true) const;

    /** \brief Apply the function \c f to each constant */
    void for_each_constant(
        std::function<void(constant_info const &d)> const &f) const;

    /** \brief Pointer equality */
    friend bool is_eqp(environment const &e1, environment const &e2) {
        return e1.raw() == e2.raw();
    }

    void display_stats() const;
};

void check_no_metavar_no_fvar(environment const &env, name const &n,
                              expr const &e);

void initialize_environment();
void finalize_environment();
}  // namespace lean
// ::::::::::::::
// equiv_manager.h
// ::::::::::::::
/*
Copyright (c) 2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <vector>

#include "kernel/expr_maps.h"

namespace lean {
```

```
class equiv_manager {
    typedef unsigned node_ref;

    struct node {
        node_ref m_parent;
        unsigned m_rank;
    };

    std::vector<node> m_nodes;
    expr_map<node_ref> m_to_node;
    bool m_use_hash;

    node_ref mk_node();
    node_ref find(node_ref n);
    void merge(node_ref n1, node_ref n2);
    node_ref to_node(expr const &e);
    bool is_equiv_core(expr const &e1, expr const &e2);

  public:
    equiv_manager() : m_use_hash(false) {}
    bool is_equiv(expr const &e1, expr const &e2, bool use_hash = false);
    void add_equiv(expr const &e1, expr const &e2);
};
}  // namespace lean
// ::::::::::::::
// expr_cache.h
// ::::::::::::::
/*
Copyright (c) 2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <vector>

#include "kernel/expr.h"

namespace lean {
/** \brief Cache for storing mappings from expressions to expressions.

    \warning The insert(k, v) method overwrites andy entry (k1, v1) when
    hash(k) == hash(k1)
*/
class expr_cache {
    struct entry {
        optional<expr> m_expr;
        expr m_result;
    };
    unsigned m_capacity;
    std::vector<entry> m_cache;
    std::vector<unsigned> m_used;

  public:
    expr_cache(unsigned c) : m_capacity(c), m_cache(c) {}
    void insert(expr const &e, expr const &v);
    expr *find(expr const &e);
    void clear();
};
}  // namespace lean
// ::::::::::::::
// expr_eq_fn.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
```

```cpp

namespace lean {
class expr;
// =======================================
// Structural equality
/** \brief Binder information is ignored in the following predicate */
bool is_equal(expr const &a, expr const &b);
inline bool operator==(expr const &a, expr const &b) { return is_equal(a, b); }
inline bool operator!=(expr const &a, expr const &b) {
    return !operator==(a, b);
}
// =======================================

/** \brief Similar to ==, but it also compares binder information */
bool is_bi_equal(expr const &a, expr const &b);
struct is_bi_equal_proc {
    bool operator()(expr const &e1, expr const &e2) const {
        return is_bi_equal(e1, e2);
    }
};

/** Similar to is_bi_equal_proc, but it has a flag that allows us to switch
 * select == or is_bi_equal */
struct is_cond_bi_equal_proc {
    bool m_use_bi;
    is_cond_bi_equal_proc(bool b) : m_use_bi(b) {}
    bool operator()(expr const &e1, expr const &e2) const {
        return m_use_bi ? is_bi_equal(e1, e2) : e1 == e2;
    }
};
}  // namespace lean
// ::::::::::::::
// expr.h
// ::::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <lean/hash.h>
#include <lean/optional.h>
#include <lean/serializer.h>
#include <lean/thread.h>

#include <algorithm>
#include <iostream>
#include <limits>
#include <string>
#include <tuple>
#include <utility>

#include "kernel/expr_eq_fn.h"
#include "kernel/level.h"
#include "util/buffer.h"
#include "util/format.h"
#include "util/kvmap.h"
#include "util/list_fn.h"
#include "util/name.h"
#include "util/nat.h"

namespace lean {
/* Binder annotations for Pi/lambda expressions */
enum class binder_info { Default, Implicit, StrictImplicit, InstImplicit, Rec };

inline binder_info mk_binder_info() { return binder_info::Default; }
inline binder_info mk_implicit_binder_info() { return binder_info::Implicit; }
inline binder_info mk_strict_implicit_binder_info() {
    return binder_info::StrictImplicit;
```

```cpp
}
inline binder_info mk_inst_implicit_binder_info() {
    return binder_info::InstImplicit;
}
inline binder_info mk_rec_info() { return binder_info::Rec; }

inline bool is_default(binder_info bi) { return bi == binder_info::Default; }
inline bool is_implicit(binder_info bi) { return bi == binder_info::Implicit; }
inline bool is_strict_implicit(binder_info bi) {
    return bi == binder_info::StrictImplicit;
}
inline bool is_inst_implicit(binder_info bi) {
    return bi == binder_info::InstImplicit;
}
inline bool is_explicit(binder_info bi) {
    return !is_implicit(bi) && !is_strict_implicit(bi) && !is_inst_implicit(bi);
}
inline bool is_rec(binder_info bi) { return bi == binder_info::Rec; }

/* Expression literal values */
enum class literal_kind { Nat, String };
class literal : public object_ref {
    explicit literal(b_obj_arg o, bool b) : object_ref(o, b) {}

   public:
    explicit literal(char const *v);
    explicit literal(unsigned v);
    explicit literal(mpz const &v);
    explicit literal(nat const &v);
    literal() : literal(0u) {}
    literal(literal const &other) : object_ref(other) {}
    literal(literal &&other) : object_ref(other) {}
    literal &operator=(literal const &other) {
        object_ref::operator=(other);
        return *this;
    }
    literal &operator=(literal &&other) {
        object_ref::operator=(other);
        return *this;
    }

    static literal_kind kind(object *o) {
        return static_cast<literal_kind>(cnstr_tag(o));
    }
    literal_kind kind() const { return kind(raw()); }
    string_ref const &get_string() const {
        lean_assert(kind() == literal_kind::String);
        return static_cast<string_ref const &>(cnstr_get_ref(*this, 0));
    }
    nat const &get_nat() const {
        lean_assert(kind() == literal_kind::Nat);
        return static_cast<nat const &>(cnstr_get_ref(*this, 0));
    }
    bool is_zero() const {
        return kind() == literal_kind::Nat && get_nat().is_zero();
    }
    friend bool operator==(literal const &a, literal const &b);
    friend bool operator<(literal const &a, literal const &b);
    void serialize(serializer &s) const { s.write_object(raw()); }
    static literal deserialize(deserializer &d) {
        return literal(d.read_object(), true);
    }
};
inline bool operator!=(literal const &a, literal const &b) { return !(a == b); }
inline serializer &operator<<(serializer &s, literal const &l) {
    l.serialize(s);
    return s;
}
inline literal read_literal(deserializer &d) { return literal::deserialize(d); }
inline deserializer &operator>>(deserializer &d, literal &l) {
```

```
    l = read_literal(d);
    return d;
}

/* ======================================
   Expressions

inductive Expr
| bvar    : Nat → Expr                                -- bound variables
| fvar    : Name → Expr                               -- free variables
| mvar    : Name → Expr                               -- meta variables
| sort    : Level → Expr                              -- Sort
| const   : Name → List Level → Expr                  -- constants
| app     : Expr → Expr → Expr                        -- application
| lam     : Name → BinderInfo → Expr → Expr → Expr    -- lambda abstraction
| forallE : Name → BinderInfo → Expr → Expr → Expr    -- (dependent) arrow
| letE    : Name → Expr → Expr → Expr → Expr          -- let expressions
| lit     : Literal → Expr                            -- literals
| mdata   : MData → Expr → Expr                       -- metadata
| proj    : Name → Nat → Expr → Expr                  -- projection
*/
enum class expr_kind {
    BVar,
    FVar,
    MVar,
    Sort,
    Const,
    App,
    Lambda,
    Pi,
    Let,
    Lit,
    MData,
    Proj
};
class expr : public object_ref {
    explicit expr(object_ref &&o) : object_ref(o) {}

    friend expr mk_lit(literal const &lit);
    friend expr mk_mdata(kvmap const &d, expr const &e);
    friend expr mk_proj(name const &s, nat const &idx, expr const &e);
    friend expr mk_bvar(nat const &idx);
    friend expr mk_mvar(name const &n);
    friend expr mk_fvar(name const &n);
    friend expr mk_const(name const &n, levels const &ls);
    friend expr mk_app(expr const &f, expr const &a);
    friend expr mk_sort(level const &l);
    friend expr mk_lambda(name const &n, expr const &t, expr const &e,
                          binder_info bi);
    friend expr mk_pi(name const &n, expr const &t, expr const &e,
                      binder_info bi);
    friend expr mk_let(name const &n, expr const &t, expr const &v,
                       expr const &b);

   public:
    expr();
    expr(expr const &other) : object_ref(other) {}
    expr(expr &&other) : object_ref(other) {}
    explicit expr(b_obj_arg o, bool b) : object_ref(o, b) {}
    explicit expr(obj_arg o) : object_ref(o) {}
    static expr_kind kind(object *o) {
        return static_cast<expr_kind>(cnstr_tag(o));
    }
    expr_kind kind() const { return kind(raw()); }

    expr &operator=(expr const &other) {
        object_ref::operator=(other);
        return *this;
    }
    expr &operator=(expr &&other) {
```

```
        object_ref::operator=(other);
        return *this;
    }

    friend bool is_eqp(expr const &e1, expr const &e2) {
        return e1.raw() == e2.raw();
    }
    void serialize(serializer &s) const { s.write_object(raw()); }
    static expr deserialize(deserializer &d) {
        return expr(d.read_object(), true);
    }
};

typedef list_ref<expr> exprs;
typedef pair<expr, expr> expr_pair;

inline serializer &operator<<(serializer &s, expr const &e) {
    e.serialize(s);
    return s;
}
inline serializer &operator<<(serializer &s, exprs const &es) {
    es.serialize(s);
    return s;
}
inline expr read_expr(deserializer &d) { return expr::deserialize(d); }
inline exprs read_exprs(deserializer &d) { return read_list_ref<expr>(d); }
inline deserializer &operator>>(deserializer &d, expr &e) {
    e = read_expr(d);
    return d;
}

inline optional<expr> none_expr() { return optional<expr>(); }
inline optional<expr> some_expr(expr const &e) { return optional<expr>(e); }
inline optional<expr> some_expr(expr &&e) {
    return optional<expr>(std::forward<expr>(e));
}

inline bool is_eqp(optional<expr> const &a, optional<expr> const &b) {
    return static_cast<bool>(a) == static_cast<bool>(b) &&
           (!a || is_eqp(*a, *b));
}

unsigned hash(expr const &e);
bool has_expr_mvar(expr const &e);
bool has_univ_mvar(expr const &e);
inline bool has_mvar(expr const &e) {
    return has_expr_mvar(e) || has_univ_mvar(e);
}
bool has_fvar(expr const &e);
bool has_univ_param(expr const &e);
unsigned get_loose_bvar_range(expr const &e);

struct expr_hash {
    unsigned operator()(expr const &e) const { return hash(e); }
};
struct expr_pair_hash {
    unsigned operator()(expr_pair const &p) const {
        return hash(hash(p.first), hash(p.second));
    }
};
struct expr_pair_eq {
    bool operator()(expr_pair const &p1, expr_pair const &p2) const {
        return p1.first == p2.first && p1.second == p2.second;
    }
};

// =======================================
// Testers
static expr_kind expr_kind_core(object *o) {
    return static_cast<expr_kind>(cnstr_tag(o));
```

```
}
inline bool is_bvar(expr const &e) { return e.kind() == expr_kind::BVar; }
inline bool is_fvar_core(object *o) {
    return expr_kind_core(o) == expr_kind::FVar;
}
inline bool is_fvar(expr const &e) { return e.kind() == expr_kind::FVar; }
inline bool is_const(expr const &e) { return e.kind() == expr_kind::Const; }
inline bool is_mvar(expr const &e) { return e.kind() == expr_kind::MVar; }
inline bool is_app(expr const &e) { return e.kind() == expr_kind::App; }
inline bool is_lambda(expr const &e) { return e.kind() == expr_kind::Lambda; }
inline bool is_pi(expr const &e) { return e.kind() == expr_kind::Pi; }
inline bool is_let(expr const &e) { return e.kind() == expr_kind::Let; }
inline bool is_sort(expr const &e) { return e.kind() == expr_kind::Sort; }
inline bool is_lit(expr const &e) { return e.kind() == expr_kind::Lit; }
inline bool is_mdata(expr const &e) { return e.kind() == expr_kind::MData; }
inline bool is_proj(expr const &e) { return e.kind() == expr_kind::Proj; }
inline bool is_binding(expr const &e) { return is_lambda(e) || is_pi(e); }

bool is_atomic(expr const &e);
bool is_arrow(expr const &t);
bool is_default_var_name(name const &n);
// =======================================

// =======================================
// Constructors
expr mk_lit(literal const &lit);
expr mk_mdata(kvmap const &d, expr const &e);
expr mk_proj(name const &s, nat const &idx, expr const &e);
inline expr mk_proj(name const &s, unsigned idx, expr const &e) {
    return mk_proj(s, nat(idx), e);
}
expr mk_bvar(nat const &idx);
inline expr mk_bvar(unsigned idx) { return mk_bvar(nat(idx)); }
expr mk_fvar(name const &n);
expr mk_const(name const &n, levels const &ls);
inline expr mk_const(name const &n) { return mk_const(n, levels()); }
expr mk_mvar(name const &n);
expr mk_app(expr const &f, expr const &a);
expr mk_app(expr const &f, unsigned num_args, expr const *args);
expr mk_app(unsigned num_args, expr const *args);
inline expr mk_app(std::initializer_list<expr> const &l) {
    return mk_app(l.size(), l.begin());
}
inline expr mk_app(buffer<expr> const &args) {
    return mk_app(args.size(), args.data());
}
inline expr mk_app(expr const &f, buffer<expr> const &args) {
    return mk_app(f, args.size(), args.data());
}
expr mk_app(expr const &f, list<expr> const &args);
inline expr mk_app(expr const &e1, expr const &e2, expr const &e3) {
    return mk_app({e1, e2, e3});
}
inline expr mk_app(expr const &e1, expr const &e2, expr const &e3,
                   expr const &e4) {
    return mk_app({e1, e2, e3, e4});
}
inline expr mk_app(expr const &e1, expr const &e2, expr const &e3,
                   expr const &e4, expr const &e5) {
    return mk_app({e1, e2, e3, e4, e5});
}
expr mk_rev_app(expr const &f, unsigned num_args, expr const *args);
expr mk_rev_app(unsigned num_args, expr const *args);
inline expr mk_rev_app(buffer<expr> const &args) {
    return mk_rev_app(args.size(), args.data());
}
inline expr mk_rev_app(expr const &f, buffer<expr> const &args) {
    return mk_rev_app(f, args.size(), args.data());
}
expr mk_lambda(name const &n, expr const &t, expr const &e,
```

```
                 binder_info bi = mk_binder_info());
expr mk_pi(name const &n, expr const &t, expr const &e,
           binder_info bi = mk_binder_info());
inline expr mk_binding(expr_kind k, name const &n, expr const &t, expr const &e,
                       binder_info bi = mk_binder_info()) {
    return k == expr_kind::Pi ? mk_pi(n, t, e, bi) : mk_lambda(n, t, e, bi);
}
expr mk_arrow(expr const &t, expr const &e);
expr mk_let(name const &n, expr const &t, expr const &v, expr const &b);
expr mk_sort(level const &l);
expr mk_Prop();
expr mk_Type();
// =======================================

// =======================================
// Accessors
inline literal const &lit_value(expr const &e) {
    lean_assert(is_lit(e));
    return static_cast<literal const &>(cnstr_get_ref(e, 0));
}
inline bool is_nat_lit(expr const &e) {
    return is_lit(e) && lit_value(e).kind() == literal_kind::Nat;
}
inline bool is_string_lit(expr const &e) {
    return is_lit(e) && lit_value(e).kind() == literal_kind::String;
}
expr lit_type(literal const &e);
inline kvmap const &mdata_data(expr const &e) {
    lean_assert(is_mdata(e));
    return static_cast<kvmap const &>(cnstr_get_ref(e, 0));
}
inline expr const &mdata_expr(expr const &e) {
    lean_assert(is_mdata(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 1));
}
inline name const &proj_sname(expr const &e) {
    lean_assert(is_proj(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
inline nat const &proj_idx(expr const &e) {
    lean_assert(is_proj(e));
    return static_cast<nat const &>(cnstr_get_ref(e, 1));
}
inline expr const &proj_expr(expr const &e) {
    lean_assert(is_proj(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 2));
}
inline nat const &bvar_idx(expr const &e) {
    lean_assert(is_bvar(e));
    return static_cast<nat const &>(cnstr_get_ref(e, 0));
}
inline bool is_bvar(expr const &e, unsigned i) {
    return is_bvar(e) && bvar_idx(e) == i;
}
inline name const &fvar_name_core(object *o) {
    lean_assert(is_fvar_core(o));
    return static_cast<name const &>(cnstr_get_ref(o, 0));
}
inline name const &fvar_name(expr const &e) {
    lean_assert(is_fvar(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
inline level const &sort_level(expr const &e) {
    lean_assert(is_sort(e));
    return static_cast<level const &>(cnstr_get_ref(e, 0));
}
inline name const &mvar_name(expr const &e) {
    lean_assert(is_mvar(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
```

```
inline name const &const_name(expr const &e) {
    lean_assert(is_const(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
inline levels const &const_levels(expr const &e) {
    lean_assert(is_const(e));
    return static_cast<levels const &>(cnstr_get_ref(e, 1));
}
inline bool is_const(expr const &e, name const &n) {
    return is_const(e) && const_name(e) == n;
}
inline expr const &app_fn(expr const &e) {
    lean_assert(is_app(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 0));
}
inline expr const &app_arg(expr const &e) {
    lean_assert(is_app(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 1));
}
inline name const &binding_name(expr const &e) {
    lean_assert(is_binding(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
inline expr const &binding_domain(expr const &e) {
    lean_assert(is_binding(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 1));
}
inline expr const &binding_body(expr const &e) {
    lean_assert(is_binding(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 2));
}
binder_info binding_info(expr const &e);
inline name const &let_name(expr const &e) {
    lean_assert(is_let(e));
    return static_cast<name const &>(cnstr_get_ref(e, 0));
}
inline expr const &let_type(expr const &e) {
    lean_assert(is_let(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 1));
}
inline expr const &let_value(expr const &e) {
    lean_assert(is_let(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 2));
}
inline expr const &let_body(expr const &e) {
    lean_assert(is_let(e));
    return static_cast<expr const &>(cnstr_get_ref(e, 3));
}
inline bool is_shared(expr const &e) { return !is_exclusive(e.raw()); }
//

// =======================================
// Update
expr update_app(expr const &e, expr const &new_fn, expr const &new_arg);
expr update_binding(expr const &e, expr const &new_domain,
                    expr const &new_body);
expr update_binding(expr const &e, expr const &new_domain, expr const &new_body,
                    binder_info bi);
expr update_sort(expr const &e, level const &new_level);
expr update_const(expr const &e, levels const &new_levels);
expr update_let(expr const &e, expr const &new_type, expr const &new_value,
                expr const &new_body);
expr update_mdata(expr const &e, expr const &new_e);
expr update_proj(expr const &e, expr const &new_e);
// =======================================

/** \brief Given \c e of the form <tt>(...(f a1) ... an)</tt>, store a1 ... an
   in args. If \c e is not an application, then nothing is stored in args.

    It returns the f. */
```

```
expr const &get_app_args(expr const &e, buffer<expr> &args);
/** \brief Similar to \c get_app_args, but stores at most num args.
    Examples:
    1) get_app_args_at_most(f a b c, 2, args);
    stores {b, c} in args and returns (f a)

    2) get_app_args_at_most(f a b c, 4, args);
    stores {a, b, c} in args and returns f */
expr const &get_app_args_at_most(expr const &e, unsigned num,
                                 buffer<expr> &args);

/** \brief Similar to \c get_app_args, but arguments are stored in reverse order
   in \c args. If e is of the form <tt>(...(f a1) ... an)</tt>, then the
   procedure stores [an, ..., a1] in \c args. */
expr const &get_app_rev_args(expr const &e, buffer<expr> &args);
/** \brief Given \c e of the form <tt>(...(f a_1) ... a_n)</tt>, return \c f. If
 * \c e is not an application, then return \c e. */
expr const &get_app_fn(expr const &e);
/** \brief Given \c e of the form <tt>(...(f a_1) ... a_n)</tt>, return \c n. If
 * \c e is not an application, then return 0. */
unsigned get_app_num_args(expr const &e);

/** \brief Return true iff \c e is a metavariable or an application of a
 * metavariable */
inline bool is_mvar_app(expr const &e) { return is_mvar(get_app_fn(e)); }

// =======================================
// Loose bound variable management

/** \brief Return true iff the given expression has loose bound variables. */
inline bool has_loose_bvars(expr const &e) {
    return get_loose_bvar_range(e) > 0;
}

/** \brief Return true iff \c e contains the loose bound variable <tt>(var
 * i)</tt>. */
bool has_loose_bvar(expr const &e, unsigned i);

/** \brief Lower the loose bound variables >= s in \c e by \c d. That is, a
   loose bound variable <tt>(var i)</tt> s.t. <tt>i >= s</tt> is mapped into
   <tt>(var i-d)</tt>.

    \pre s >= d */
expr lower_loose_bvars(expr const &e, unsigned s, unsigned d);
expr lower_loose_bvars(expr const &e, unsigned d);

/** \brief Lift loose bound variables >= s in \c e by d. */
expr lift_loose_bvars(expr const &e, unsigned s, unsigned d);
expr lift_loose_bvars(expr const &e, unsigned d);
// =======================================

// =======================================
// Implicit argument inference
/**
   \brief Given \c t of the form <tt>Pi (x_1 : A_1) ... (x_k : A_k), B</tt>,
   mark the first \c num_params as implicit if they are not already marked, and
   they occur in the remaining arguments. If \c strict is false, then we
   also mark it implicit if it occurs in \c B.
*/
expr infer_implicit(expr const &t, unsigned num_params, bool strict);
expr infer_implicit(expr const &t, bool strict);
// =======================================

// =======================================
// Low level (raw) printing
std::ostream &operator<<(std::ostream &out, expr const &e);
// =======================================

void initialize_expr();
void finalize_expr();
```

```

/* ================= LEGACY ============== */
inline bool has_expr_metavar(expr const &e) { return has_expr_mvar(e); }
inline bool has_univ_metavar(expr const &e) { return has_univ_mvar(e); }
inline bool has_metavar(expr const &e) { return has_mvar(e); }
inline bool has_param_univ(expr const &e) { return has_univ_param(e); }
inline bool is_var(expr const &e) { return is_bvar(e); }
inline bool is_var(expr const &e, unsigned idx) { return is_bvar(e, idx); }
inline bool is_metavar(expr const &e) { return is_mvar(e); }
inline bool is_metavar_app(expr const &e) { return is_mvar_app(e); }
inline expr mk_metavar(name const &n) { return mk_mvar(n); }
inline expr mk_constant(name const &n, levels const &ls) {
    return mk_const(n, ls);
}
inline expr mk_constant(name const &n) { return mk_constant(n, levels()); }
inline bool is_constant(expr const &e) { return is_const(e); }
inline expr update_constant(expr const &e, levels const &new_levels) {
    return update_const(e, new_levels);
}
/** \brief Similar to \c has_expr_metavar, but ignores metavariables occurring
   in local constant types.
    It also returns the meta-variable application found in \c e. */
optional<expr> has_expr_metavar_strict(expr const &e);
inline bool is_constant(expr const &e, name const &n) { return is_const(e, n); }
}  // namespace lean
// ::::::::::::::
// expr_maps.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <functional>
#include <unordered_map>

#include "kernel/expr.h"

namespace lean {
// Maps based on structural equality. That is, two keys are equal iff they are
// structurally equal
template <typename T>
using expr_map =
    typename std::unordered_map<expr, T, expr_hash, std::equal_to<expr>>;
// The following map also takes into account binder information
template <typename T>
using expr_bi_map =
    typename std::unordered_map<expr, T, expr_hash, is_bi_equal_proc>;

template <typename T>
class expr_cond_bi_map
    : public std::unordered_map<expr, T, expr_hash, is_cond_bi_equal_proc> {
   public:
    expr_cond_bi_map(bool use_bi = false)
        : std::unordered_map<expr, T, expr_hash, is_cond_bi_equal_proc>(
              10, expr_hash(), is_cond_bi_equal_proc(use_bi)) {}
};
};  // namespace lean
// ::::::::::::::
// expr_sets.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
```

```
#include <lean/hash.h>

#include <functional>
#include <unordered_set>
#include <utility>

#include "kernel/expr.h"

namespace lean {
typedef std::unordered_set<expr, expr_hash, std::equal_to<expr>> expr_set;
}
// ::::::::::::::
// find_fn.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/expr.h"
#include "kernel/for_each_fn.h"

namespace lean {
/** \brief Return a subexpression of \c e that satisfies the predicate \c p. */
template <typename P>
optional<expr> find(expr const &e, P p) {
    optional<expr> result;
    for_each(e, [&](expr const &e, unsigned offset) {
        if (result) {
            return false;
        } else if (p(e, offset)) {
            result = e;
            return false;
        } else {
            return true;
        }
    });
    return result;
}
}  // namespace lean
// ::::::::::::::
// for_each_fn.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <functional>
#include <memory>
#include <utility>

#include "kernel/expr.h"
#include "kernel/expr_sets.h"
#include "util/buffer.h"

namespace lean {
/** \brief Expression visitor.

    The argument \c f must be a lambda (function object) containing the method

    <code>
    bool operator()(expr const & e, unsigned offset)
    </code>

    The \c offset is the number of binders under which \c e occurs.
```

```
*/
void for_each(expr const &e,
              std::function<bool(expr const &, unsigned)> &&f);  // NOLINT
}  // namespace lean
// ::::::::::::::
// inductive.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
#include "kernel/instantiate.h"
namespace lean {
/**\ brief Return recursor name for the given inductive datatype name */
name mk_rec_name(name const &I);

/* Auxiliary function for to_cnstr_when_K */
optional<expr> mk_nullary_cnstr(environment const &env, expr const &type,
                                unsigned num_params);

/* For datatypes that support K-axiom, given `e` an element of that type, we
   convert (if possible) to the default constructor. For example, if `e : a =
   a`, then this method returns `eq.refl a` */
template <typename WHNF, typename INFER, typename IS_DEF_EQ>
inline optional<expr> to_cnstr_when_K(environment const &env,
                                      recursor_val const &rval, expr const &e,
                                      WHNF const &whnf, INFER const &infer_type,
                                      IS_DEF_EQ const &is_def_eq) {
    lean_assert(rval.is_k());
    expr app_type = whnf(infer_type(e));
    expr const &app_type_I = get_app_fn(app_type);
    if (!is_constant(app_type_I) || const_name(app_type_I) != rval.get_induct())
        return none_expr();  // type incorrect
    if (has_expr_mvar(app_type)) {
        buffer<expr> app_type_args;
        get_app_args(app_type, app_type_args);
        for (unsigned i = rval.get_nparams(); i < app_type_args.size(); i++) {
            if (has_expr_metavar(app_type_args[i])) return none_expr();
        }
    }
    optional<expr> new_cnstr_app =
        mk_nullary_cnstr(env, app_type, rval.get_nparams());
    if (!new_cnstr_app) return none_expr();
    expr new_type = infer_type(*new_cnstr_app);
    if (!is_def_eq(app_type, new_type)) return none_expr();
    return some_expr(*new_cnstr_app);
}

optional<recursor_rule> get_rec_rule_for(recursor_val const &rec_val,
                                         expr const &major);

expr nat_lit_to_constructor(expr const &e);
expr string_lit_to_constructor(expr const &e);

template <typename WHNF, typename INFER, typename IS_DEF_EQ>
inline optional<expr> inductive_reduce_rec(environment const &env,
                                           expr const &e, WHNF const &whnf,
                                           INFER const &infer_type,
                                           IS_DEF_EQ const &is_def_eq) {
    expr const &rec_fn = get_app_fn(e);
    if (!is_constant(rec_fn)) return none_expr();
    optional<constant_info> rec_info = env.find(const_name(rec_fn));
    if (!rec_info || !rec_info->is_recursor()) return none_expr();
    buffer<expr> rec_args;
    get_app_args(e, rec_args);
    recursor_val const &rec_val = rec_info->to_recursor_val();
```

```
    unsigned major_idx = rec_val.get_major_idx();
    if (major_idx >= rec_args.size())
        return none_expr();  // major premise is missing
    expr major = rec_args[major_idx];
    if (rec_val.is_k()) {
        if (optional<expr> c = to_cnstr_when_K(env, rec_val, major, whnf,
                                               infer_type, is_def_eq)) {
            major = *c;
        }
    }
    major = whnf(major);
    if (is_nat_lit(major)) major = nat_lit_to_constructor(major);
    if (is_string_lit(major)) major = string_lit_to_constructor(major);
    optional<recursor_rule> rule = get_rec_rule_for(rec_val, major);
    if (!rule) return none_expr();
    buffer<expr> major_args;
    get_app_args(major, major_args);
    if (rule->get_nfields() > major_args.size()) return none_expr();
    if (length(const_levels(rec_fn)) != length(rec_info->get_lparams()))
        return none_expr();
    expr rhs = instantiate_lparams(rule->get_rhs(), rec_info->get_lparams(),
                                   const_levels(rec_fn));
    /* apply parameters, motives and minor premises from recursor application.
     */
    rhs = mk_app(
        rhs,
        rec_val.get_nparams() + rec_val.get_nmotives() + rec_val.get_nminors(),
        rec_args.data());
    /* The number of parameters in the constructor is not necessarily
       equal to the number of parameters in the recursor when we have
       nested inductive types. */
    unsigned nparams = major_args.size() - rule->get_nfields();
    /* apply fields from major premise */
    rhs = mk_app(rhs, rule->get_nfields(), major_args.data() + nparams);
    if (rec_args.size() > major_idx + 1) {
        /* recursor application has more arguments after major premise */
        unsigned nextra = rec_args.size() - major_idx - 1;
        rhs = mk_app(rhs, nextra, rec_args.data() + major_idx + 1);
    }
    return some_expr(rhs);
}

template <typename WHNF, typename IS_STUCK>
optional<expr> inductive_is_stuck(environment const &env, expr const &e,
                                  WHNF const &whnf, IS_STUCK const &is_stuck) {
    expr const &rec_fn = get_app_fn(e);
    if (!is_constant(rec_fn)) return none_expr();
    optional<constant_info> rec_info = env.find(const_name(rec_fn));
    if (!rec_info || !rec_info->is_recursor()) return none_expr();
    buffer<expr> rec_args;
    get_app_args(e, rec_args);
    recursor_val const &rec_val = rec_info->to_recursor_val();
    unsigned major_idx = rec_val.get_major_idx();
    if (rec_args.size() < major_idx + 1) return none_expr();
    expr cnstr_app = whnf(rec_args[major_idx]);
    if (rec_val.is_k()) {
        /* TODO(Leo): make it more precise.  Remark: this piece of
           code does not affect the correctness of the kernel, but the
           effectiveness of the elaborator. */
        return none_expr();
    } else {
        return is_stuck(cnstr_app);
    }
}

void initialize_inductive();
void finalize_inductive();
}  // namespace lean
// ::::::::::::::
// init_module.h
```

```
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
namespace lean {
void initialize_kernel_module();
void finalize_kernel_module();
}  // namespace lean
// ::::::::::::::
// instantiate.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <functional>

#include "kernel/expr.h"

namespace lean {
class ro_metavar_env;
/** \brief Replace the loose bound variables with indices 0, ..., n-1 with s[0],
 * ..., s[n-1] in e. */
expr instantiate(expr const &e, unsigned n, expr const *s);
expr instantiate(expr const &e, std::initializer_list<expr> const &l);
/** \brief Replace loose bound variable \c i with \c s in \c e. */
expr instantiate(expr const &e, unsigned i, expr const &s);
/** \brief Replace loose bound variable \c 0 with \c s in \c e. */
expr instantiate(expr const &e, expr const &s);

/** \brief Replace the free variables with indices 0, ..., n-1 with s[n-1], ...,
 * s[0] in e. */
expr instantiate_rev(expr const &e, unsigned n, expr const *s);
inline expr instantiate_rev(expr const &e, buffer<expr> const &s) {
    return instantiate_rev(e, s.size(), s.data());
}

expr apply_beta(expr f, unsigned num_rev_args, expr const *rev_args);
bool is_head_beta(expr const &t);
expr head_beta_reduce(expr const &t);
/* If `e` is of the form `(fun x, t) a` return `head_beta_const_fn(t)` if `t`
   does not depend on `x`,
   and `e` otherwise. We also reduce `(fun x_1 ... x_n, x_i) a_1 ... a_n` into
   `a_[n-i-1]` */
expr cheap_beta_reduce(expr const &e);

/** \brief Instantiate the universe level parameters \c ps occurring in \c e
   with the levels \c ls. \pre length(ps) == length(ls) */
expr instantiate_lparams(expr const &e, names const &ps, levels const &ls);

class constant_info;
/** \brief Instantiate the universe level parameters of the type of the given
   constant. \pre d.get_num_lparams() == length(ls) */
expr instantiate_type_lparams(constant_info const &info, levels const &ls);
/** \brief Instantiate the universe level parameters of the value of the given
   constant. \pre d.get_num_lparams() == length(ls) */
expr instantiate_value_lparams(constant_info const &info, levels const &ls);
}  // namespace lean
// ::::::::::::::
// kernel_exception.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
#include "kernel/local_ctx.h"

namespace lean {
/** \brief Base class for all kernel exceptions. */
class kernel_exception : public exception {
   protected:
    environment m_env;

   public:
    kernel_exception(environment const &env)
        : exception("kernel exception"), m_env(env) {}
    kernel_exception(environment const &env, char const *msg)
        : exception(msg), m_env(env) {}
    kernel_exception(environment const &env, sstream const &strm)
        : exception(strm), m_env(env) {}
    environment const &get_environment() const { return m_env; }
    environment const &env() const { return m_env; }
};

class unknown_constant_exception : public kernel_exception {
    name m_name;

   public:
    unknown_constant_exception(environment const &env, name const &n)
        : kernel_exception(env), m_name(n) {}
    name const &get_name() const { return m_name; }
};

class already_declared_exception : public kernel_exception {
    name m_name;

   public:
    already_declared_exception(environment const &env, name const &n)
        : kernel_exception(env), m_name(n) {}
    name const &get_name() const { return m_name; }
};

class definition_type_mismatch_exception : public kernel_exception {
    declaration m_decl;
    expr m_given_type;

   public:
    definition_type_mismatch_exception(environment const &env,
                                       declaration const &decl,
                                       expr const &given_type)
        : kernel_exception(env), m_decl(decl), m_given_type(given_type) {}
    declaration const &get_declaration() const { return m_decl; }
    expr const &get_given_type() const { return m_given_type; }
};

class declaration_has_metavars_exception : public kernel_exception {
    name m_name;
    expr m_expr;

   public:
    declaration_has_metavars_exception(environment const &env, name const &n,
                                       expr const &e)
        : kernel_exception(env), m_name(n), m_expr(e) {}
    name const &get_decl_name() const { return m_name; }
    expr const &get_expr() const { return m_expr; }
};

class declaration_has_free_vars_exception : public kernel_exception {
    name m_name;
```

```
    expr m_expr;

  public:
    declaration_has_free_vars_exception(environment const &env, name const &n,
                                        expr const &e)
        : kernel_exception(env), m_name(n), m_expr(e) {}
    name const &get_decl_name() const { return m_name; }
    expr const &get_expr() const { return m_expr; }
};

class kernel_exception_with_lctx : public kernel_exception {
    local_ctx m_lctx;

  public:
    kernel_exception_with_lctx(environment const &env, local_ctx const &lctx)
        : kernel_exception(env), m_lctx(lctx) {}
    local_ctx const &get_local_ctx() const { return m_lctx; }
};

class function_expected_exception : public kernel_exception_with_lctx {
    expr m_fn;

  public:
    function_expected_exception(environment const &env, local_ctx const &lctx,
                                expr const &fn)
        : kernel_exception_with_lctx(env, lctx), m_fn(fn) {}
    expr const &get_fn() const { return m_fn; }
};

class type_expected_exception : public kernel_exception_with_lctx {
    expr m_type;

  public:
    type_expected_exception(environment const &env, local_ctx const &lctx,
                            expr const &type)
        : kernel_exception_with_lctx(env, lctx), m_type(type) {}
    expr const &get_type() const { return m_type; }
};

class type_mismatch_exception : public kernel_exception_with_lctx {
    expr m_given_type;
    expr m_expected_type;

  public:
    type_mismatch_exception(environment const &env, local_ctx const &lctx,
                            expr const &given_type, expr const &expected_type)
        : kernel_exception_with_lctx(env, lctx),
          m_given_type(given_type),
          m_expected_type(expected_type) {}
    expr const &get_given_type() const { return m_given_type; }
    expr const &get_expected_type() const { return m_expected_type; }
};

class def_type_mismatch_exception : public type_mismatch_exception {
    name m_name;

  public:
    def_type_mismatch_exception(environment const &env, local_ctx const &lctx,
                                name const &n, expr const &given_type,
                                expr const &expected_type)
        : type_mismatch_exception(env, lctx, given_type, expected_type),
          m_name(n) {}
    name const &get_name() const { return m_name; }
};

class expr_type_mismatch_exception : public kernel_exception_with_lctx {
    expr m_expr;
    expr m_expected_type;

  public:
```

````
    expr_type_mismatch_exception(environment const &env, local_ctx const &lctx,
                                 expr const &e, expr const &expected_type)
        : kernel_exception_with_lctx(env, lctx),
          m_expr(e),
          m_expected_type(expected_type) {}
    expr const &get_expr() const { return m_expr; }
    expr const &get_expected_type() const { return m_expected_type; }
};

class app_type_mismatch_exception : public kernel_exception_with_lctx {
    expr m_app;
    expr m_function_type;
    expr m_arg_type;

   public:
    app_type_mismatch_exception(environment const &env, local_ctx const &lctx,
                                expr const &app, expr const &function_type,
                                expr const &arg_type)
        : kernel_exception_with_lctx(env, lctx),
          m_app(app),
          m_function_type(function_type),
          m_arg_type(arg_type) {}
    expr const &get_app() const { return m_app; }
    expr const &get_function_type() const { return m_function_type; }
    expr const &get_arg_type() const { return m_arg_type; }
};

class invalid_proj_exception : public kernel_exception_with_lctx {
    expr m_proj;

   public:
    invalid_proj_exception(environment const &env, local_ctx const &lctx,
                           expr const &proj)
        : kernel_exception_with_lctx(env, lctx), m_proj(proj) {}
    expr const &get_proj() const { return m_proj; }
};

/*
Helper function for interfacing C++ code with code written in Lean.
It executes closure `f` which produces an object_ref of type `A` and may throw
an `kernel_exception` or `exception`. Then, convert result into `Except
KernelException T` where `T` is the type of the lean objected represented by
`A`. We use the constructor `KernelException.other <msg>` to handle C++
`exception` objects which are not `kernel_exception`.
```
inductive KernelException
0  | unknownConstant  (env : Environment) (name : Name)
1  | alreadyDeclared  (env : Environment) (name : Name)
2  | declTypeMismatch (env : Environment) (decl : Declaration) (givenType :
Expr) 3  | declHasMVars     (env : Environment) (name : Name) (expr : Expr) 4  |
declHasFVars     (env : Environment) (name : Name) (expr : Expr) 5  |
funExpected      (env : Environment) (lctx : LocalContext) (expr : Expr) 6  |
typeExpected     (env : Environment) (lctx : LocalContext) (expr : Expr) 7  |
letTypeMismatch  (env : Environment) (lctx : LocalContext) (name : Name)
(givenType : Expr) (expectedType : Expr) 8  | exprTypeMismatch (env :
Environment) (lctx : LocalContext) (expr : Expr) (expectedType : Expr) 9  |
appTypeMismatch  (env : Environment) (lctx : LocalContext) (app : Expr) (funType
: Expr) (argType : Expr) 10 | invalidProj      (env : Environment) (lctx :
LocalContext) (proj : Expr) 11 | other            (msg : String)
```
*/
template <typename A>
object *catch_kernel_exceptions(std::function<A()> const &f) {
    try {
        A a = f();
        return mk_cnstr(1, a).steal();
    } catch (unknown_constant_exception &ex) {
        // 0  | unknownConstant  (env : Environment) (name : Name)
        return mk_cnstr(0, mk_cnstr(0, ex.env(), ex.get_name())).steal();
    } catch (already_declared_exception &ex) {
````

```
        // 1  | alreadyDeclared  (env : Environment) (name : Name)
        return mk_cnstr(0, mk_cnstr(1, ex.env(), ex.get_name())).steal();
    } catch (definition_type_mismatch_exception &ex) {
        // 2  | declTypeMismatch (env : Environment) (decl : Declaration)
        // (givenType : Expr)
        return mk_cnstr(0, mk_cnstr(2, ex.env(), ex.get_declaration(),
                                    ex.get_given_type()))
            .steal();
    } catch (declaration_has_metavars_exception &ex) {
        // 3  | declHasMVars     (env : Environment) (name : Name) (expr : Expr)
        return mk_cnstr(
                   0, mk_cnstr(3, ex.env(), ex.get_decl_name(), ex.get_expr()))
            .steal();
    } catch (declaration_has_free_vars_exception &ex) {
        // 4  | declHasFVars     (env : Environment) (name : Name) (expr : Expr)
        return mk_cnstr(
                   0, mk_cnstr(4, ex.env(), ex.get_decl_name(), ex.get_expr()))
            .steal();
    } catch (function_expected_exception &ex) {
        // 5  | funExpected      (env : Environment) (lctx : LocalContext) (expr
        // : Expr)
        return mk_cnstr(0,
                        mk_cnstr(5, ex.env(), ex.get_local_ctx(), ex.get_fn()))
            .steal();
    } catch (type_expected_exception &ex) {
        // 6  | typeExpected     (env : Environment) (lctx : LocalContext) (expr
        // : Expr)
        return mk_cnstr(
                   0, mk_cnstr(6, ex.env(), ex.get_local_ctx(), ex.get_type()))
            .steal();
    } catch (def_type_mismatch_exception &ex) {
        // 7  | letTypeMismatch  (env : Environment) (lctx : LocalContext) (name
        // : Name) (givenType : Expr) (expectedType : Expr)
        return mk_cnstr(0,
                        mk_cnstr(7, ex.env(), ex.get_local_ctx(), ex.get_name(),
                                 ex.get_given_type(), ex.get_expected_type()))
            .steal();
    } catch (expr_type_mismatch_exception &ex) {
        // 8  | exprTypeMismatch (env : Environment) (lctx : LocalContext) (expr
        // : Expr) (expectedType : Expr)
        return mk_cnstr(0, mk_cnstr(8, ex.env(), ex.get_local_ctx(),
                                    ex.get_expr(), ex.get_expected_type()))
            .steal();
    } catch (app_type_mismatch_exception &ex) {
        // 9  | appTypeMismatch  (env : Environment) (lctx : LocalContext) (app
        // : Expr) (funType : Expr) (argType : Expr)
        return mk_cnstr(0,
                        mk_cnstr(9, ex.env(), ex.get_local_ctx(), ex.get_app(),
                                 ex.get_function_type(), ex.get_arg_type()))
            .steal();
    } catch (invalid_proj_exception &ex) {
        // 10 | invalidProj      (env : Environment) (lctx : LocalContext) (proj
        // : Expr)
        return mk_cnstr(
                   0, mk_cnstr(10, ex.env(), ex.get_local_ctx(), ex.get_proj()))
            .steal();
    } catch (exception &ex) {
        // 11 | other            (msg : String)
        return mk_cnstr(0, mk_cnstr(11, string_ref(ex.what()))).steal();
    }
}
}  // namespace lean
// ::::::::::::
// level.h
// ::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
```

```
*/
#pragma once
#include <lean/optional.h>

#include <algorithm>
#include <iostream>
#include <utility>

#include "util/format.h"
#include "util/list_ref.h"
#include "util/name.h"
#include "util/options.h"

namespace lean {
class environment;
struct level_cell;
/**
inductive level
| zero   : level
| succ   : level → level
| max    : level → level → level
| imax   : level → level → level
| param  : name → level
| mvar   : name → level

We level.imax to handle Pi-types.
*/
enum class level_kind { Zero, Succ, Max, IMax, Param, MVar };

/** \brief Universe level. */
class level : public object_ref {
    friend level mk_succ(level const &l);
    friend level mk_max_core(level const &l1, level const &l2);
    friend level mk_imax_core(level const &l1, level const &l2);
    friend level mk_univ_param(name const &n);
    friend level mk_univ_mvar(name const &n);
    explicit level(object_ref &&o) : object_ref(o) {}

   public:
    /** \brief Universe zero */
    level();
    explicit level(obj_arg o) : object_ref(o) {}
    explicit level(b_obj_arg o, bool b) : object_ref(o, b) {}
    level(level const &other) : object_ref(other) {}
    level(level &&other) : object_ref(other) {}
    level_kind kind() const {
        return static_cast<level_kind>(lean_ptr_tag(raw()));
    }
    unsigned hash() const;

    level &operator=(level const &other) {
        object_ref::operator=(other);
        return *this;
    }
    level &operator=(level &&other) {
        object_ref::operator=(other);
        return *this;
    }

    friend bool is_eqp(level const &l1, level const &l2) {
        return l1.raw() == l2.raw();
    }
    void serialize(serializer &s) const { s.write_object(raw()); }
    static level deserialize(deserializer &d) {
        return level(d.read_object(), true);
    }

    bool is_zero() const { return kind() == level_kind::Zero; }
    bool is_succ() const { return kind() == level_kind::Succ; }
    bool is_max() const { return kind() == level_kind::Max; }
```

```
    bool is_imax() const { return kind() == level_kind::IMax; }
    bool is_param() const { return kind() == level_kind::Param; }
    bool is_mvar() const { return kind() == level_kind::MVar; }

    friend inline level const &max_lhs(level const &l) {
        lean_assert(l.is_max());
        return static_cast<level const &>(cnstr_get_ref(l, 0));
    }
    friend inline level const &max_rhs(level const &l) {
        lean_assert(l.is_max());
        return static_cast<level const &>(cnstr_get_ref(l, 1));
    }
    friend inline level const &imax_lhs(level const &l) {
        lean_assert(l.is_imax());
        return static_cast<level const &>(cnstr_get_ref(l, 0));
    }
    friend inline level const &imax_rhs(level const &l) {
        lean_assert(l.is_imax());
        return static_cast<level const &>(cnstr_get_ref(l, 1));
    }
    friend inline level const &level_lhs(level const &l) {
        lean_assert(l.is_max() || l.is_imax());
        return static_cast<level const &>(cnstr_get_ref(l, 0));
    }
    friend inline level const &level_rhs(level const &l) {
        lean_assert(l.is_max() || l.is_imax());
        return static_cast<level const &>(cnstr_get_ref(l, 1));
    }
    friend inline level const &succ_of(level const &l) {
        lean_assert(l.is_succ());
        return static_cast<level const &>(cnstr_get_ref(l, 0));
    }
    friend inline name const &param_id(level const &l) {
        lean_assert(l.is_param());
        return static_cast<name const &>(cnstr_get_ref(l, 0));
    }
    friend inline name const &mvar_id(level const &l) {
        lean_assert(l.is_mvar());
        return static_cast<name const &>(cnstr_get_ref(l, 0));
    }
    friend inline name const &level_id(level const &l) {
        lean_assert(l.is_param() || l.is_mvar());
        return static_cast<name const &>(cnstr_get_ref(l, 0));
    }
};

typedef list_ref<level> levels;
typedef pair<level, level> level_pair;

bool operator==(level const &l1, level const &l2);
inline bool operator!=(level const &l1, level const &l2) {
    return !operator==(l1, l2);
}

struct level_hash {
    unsigned operator()(level const &n) const { return n.hash(); }
};
struct level_eq {
    bool operator()(level const &n1, level const &n2) const { return n1 == n2; }
};

inline serializer &operator<<(serializer &s, level const &l) {
    l.serialize(s);
    return s;
}
inline serializer &operator<<(serializer &s, levels const &ls) {
    ls.serialize(s);
    return s;
}
inline level read_level(deserializer &d) { return level::deserialize(d); }
```

```
inline levels read_levels(deserializer &d) { return read_list_ref<level>(d); }
inline deserializer &operator>>(deserializer &d, level &l) {
    l = read_level(d);
    return d;
}

inline optional<level> none_level() { return optional<level>(); }
inline optional<level> some_level(level const &e) { return optional<level>(e); }
inline optional<level> some_level(level &&e) {
    return optional<level>(std::forward<level>(e));
}

level const &mk_level_zero();
level const &mk_level_one();
level mk_max_core(level const &l1, level const &l2);
level mk_imax_core(level const &l1, level const &l2);
level mk_max(level const &l1, level const &l2);
level mk_imax(level const &l1, level const &l2);
level mk_succ(level const &l);
level mk_univ_param(name const &n);
level mk_univ_mvar(name const &n);

/** \brief Convert (succ^k l) into (l, k). If l is not a succ, then return (l,
 * 0) */
pair<level, unsigned> to_offset(level l);

inline unsigned hash(level const &l) { return l.hash(); }
inline level_kind kind(level const &l) { return l.kind(); }
inline bool is_zero(level const &l) { return l.is_zero(); }
inline bool is_param(level const &l) { return l.is_param(); }
inline bool is_mvar(level const &l) { return l.is_mvar(); }
inline bool is_succ(level const &l) { return l.is_succ(); }
inline bool is_max(level const &l) { return l.is_max(); }
inline bool is_imax(level const &l) { return l.is_imax(); }
bool is_one(level const &l);

unsigned get_depth(level const &l);

/** \brief Return true iff \c l is an explicit level.
    We say a level l is explicit iff
    1) l is zero OR
    2) l = succ(l') and l' is explicit */
bool is_explicit(level const &l);
/** \brief Convert an explicit universe into a unsigned integer.
    \pre is_explicit(l) */
unsigned to_explicit(level const &l);
/** \brief Return true iff \c l contains placeholder (aka meta parameters). */
bool has_mvar(level const &l);
/** \brief Return true iff \c l contains parameters */
bool has_param(level const &l);

/** \brief Return a new level expression based on <tt>l == succ(arg)</tt>, where
   \c arg is replaced with \c new_arg. \pre is_succ(l) */
level update_succ(level const &l, level const &new_arg);
/** \brief Return a new level expression based on <tt>l == max(lhs, rhs)</tt>,
   where \c lhs is replaced with \c new_lhs and \c rhs is replaced with \c
   new_rhs.

    \pre is_max(l) || is_imax(l) */
level update_max(level const &l, level const &new_lhs, level const &new_rhs);

/** \brief Return true if lhs and rhs denote the same level.
    The check is done by normalization. */
bool is_equivalent(level const &lhs, level const &rhs);
/** \brief Return the given level expression normal form */
level normalize(level const &l);

/** \brief If the result is true, then forall assignments \c A that assigns all
   parameters and metavariables occuring in \c l1 and \l2, we have that the
   universe level l1[A] is bigger or equal to l2[A].
```

```cpp

    \remark This function assumes l1 and l2 are normalized */
bool is_geq_core(level l1, level l2);

bool is_geq(level const &l1, level const &l2);

bool levels_has_mvar(object *ls);
bool has_mvar(levels const &ls);
bool levels_has_param(object *ls);
bool has_param(levels const &ls);

/** \brief An arbitrary (monotonic) total order on universe level terms. */
bool is_lt(level const &l1, level const &l2, bool use_hash);
bool is_lt(levels const &as, levels const &bs, bool use_hash);
struct level_quick_cmp {
    int operator()(level const &l1, level const &l2) const {
        return is_lt(l1, l2, true) ? -1 : (l1 == l2 ? 0 : 1);
    }
};

/** \brief Functional for applying <tt>F</tt> to each level expressions. */
class for_each_level_fn {
    std::function<bool(level const &)> m_f;  // NOLINT
    void apply(level const &l);

   public:
    template <typename F>
    for_each_level_fn(F const &f) : m_f(f) {}
    void operator()(level const &l) { return apply(l); }
};
template <typename F>
void for_each(level const &l, F const &f) {
    return for_each_level_fn(f)(l);
}

/** \brief Functional for applying <tt>F</tt> to the level expressions. */
class replace_level_fn {
    std::function<optional<level>(level const &)> m_f;
    level apply(level const &l);

   public:
    template <typename F>
    replace_level_fn(F const &f) : m_f(f) {}
    level operator()(level const &l) { return apply(l); }
};
template <typename F>
level replace(level const &l, F const &f) {
    return replace_level_fn(f)(l);
}

/** \brief Return true if \c u occurs in \c l */
bool occurs(level const &u, level const &l);

/** \brief If \c l contains a parameter that is not in \c ps, then return it.
 * Otherwise, return none. */
optional<name> get_undef_param(level const &l, names const &lparams);

/** \brief Instantiate the universe level parameters \c ps occurring in \c l
   with the levels \c ls. \pre length(ps) == length(ls) */
level instantiate(level const &l, names const &ps, levels const &ls);

/** \brief Printer for debugging purposes */
std::ostream &operator<<(std::ostream &out, level const &l);

/** \brief If the result is true, then forall assignments \c A that assigns all
   parameters and metavariables occuring in \c l, l[A] != zero. */
bool is_not_zero(level const &l);

/** \brief Pretty print the given level expression, unicode characters are used
 * if \c unicode is \c true. */
```

```
format pp(level l, bool unicode, unsigned indent);
/** \brief Pretty print the given level expression using the given configuration
 * options. */
format pp(level const &l, options const &opts = options());

/** \brief Pretty print lhs <= rhs, unicode characters are used if \c unicode is
 * \c true. */
format pp(level const &lhs, level const &rhs, bool unicode, unsigned indent);
/** \brief Pretty print lhs <= rhs using the given configuration options. */
format pp(level const &lhs, level const &rhs, options const &opts = options());
/** \brief Convert a list of universe level parameter names into a list of
 * levels. */
levels lparams_to_levels(names const &ps);

void initialize_level();
void finalize_level();
}  // namespace lean
void print(lean::level const &l);
// ::::::::::::
// local_ctx.h
// ::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/expr.h"
#include "util/name_generator.h"
#include "util/name_map.h"
#include "util/rb_map.h"

namespace lean {
/*
inductive LocalDecl
| cdecl (index : Nat) (name : Name) (userName : Name) (type : Expr) (bi :
BinderInfo) | ldecl (index : Nat) (name : Name) (userName : Name) (type : Expr)
(value : Expr)
*/
class local_decl : public object_ref {
    friend class local_ctx;
    friend class local_context;
    friend void initialize_local_ctx();
    local_decl(unsigned idx, name const &n, name const &un, expr const &t,
               expr const &v);
    local_decl(local_decl const &d, expr const &t, expr const &v);
    local_decl(unsigned idx, name const &n, name const &un, expr const &t,
               binder_info bi);
    local_decl(local_decl const &d, expr const &t);

   public:
    local_decl();
    local_decl(local_decl const &other) : object_ref(other) {}
    local_decl(local_decl &&other) : object_ref(other) {}
    local_decl(obj_arg o) : object_ref(o) {}
    local_decl(b_obj_arg o, bool) : object_ref(o, true) {}
    local_decl &operator=(local_decl const &other) {
        object_ref::operator=(other);
        return *this;
    }
    local_decl &operator=(local_decl &&other) {
        object_ref::operator=(other);
        return *this;
    }
    friend bool is_eqp(local_decl const &d1, local_decl const &d2) {
        return d1.raw() == d2.raw();
    }
    unsigned get_idx() const {
        return static_cast<nat const &>(cstr_get_ref(raw(), 0))
```

```
            .get_small_value();
    }
    name const &get_name() const {
        return static_cast<name const &>(cnstr_get_ref(raw(), 1));
    }
    name const &get_user_name() const {
        return static_cast<name const &>(cnstr_get_ref(raw(), 2));
    }
    expr const &get_type() const {
        return static_cast<expr const &>(cnstr_get_ref(raw(), 3));
    }
    optional<expr> get_value() const {
        if (cnstr_tag(raw()) == 0) return none_expr();
        return some_expr(static_cast<expr const &>(cnstr_get_ref(raw(), 4)));
    }
    binder_info get_info() const;
    expr mk_ref() const;
};

/* Plain local context object used by the kernel type checker. */
class local_ctx : public object_ref {
   protected:
    template <bool is_lambda>
    expr mk_binding(unsigned num, expr const *fvars, expr const &b,
                    bool remove_dead_let = false) const;

   public:
    local_ctx();
    explicit local_ctx(obj_arg o) : object_ref(o) {}
    local_ctx(b_obj_arg o, bool) : object_ref(o, true) {}
    local_ctx(local_ctx const &other) : object_ref(other) {}
    local_ctx(local_ctx &&other) : object_ref(other) {}
    local_ctx &operator=(local_ctx const &other) {
        object_ref::operator=(other);
        return *this;
    }
    local_ctx &operator=(local_ctx &&other) {
        object_ref::operator=(other);
        return *this;
    }

    bool empty() const;

    /* Low level `mk_local_decl` */
    local_decl mk_local_decl(name const &n, name const &un, expr const &type,
                             binder_info bi);
    /* Low level `mk_local_decl` */
    local_decl mk_local_decl(name const &n, name const &un, expr const &type,
                             expr const &value);

    expr mk_local_decl(name_generator &g, name const &un, expr const &type,
                       binder_info bi = mk_binder_info()) {
        return mk_local_decl(g.next(), un, type, bi).mk_ref();
    }

    expr mk_local_decl(name_generator &g, name const &un, expr const &type,
                       expr const &value) {
        return mk_local_decl(g.next(), un, type, value).mk_ref();
    }

    /** \brief Return the local declarations for the given reference. */
    optional<local_decl> find_local_decl(name const &n) const;
    optional<local_decl> find_local_decl(expr const &e) const {
        return find_local_decl(fvar_name(e));
    }

    local_decl get_local_decl(name const &n) const;
    local_decl get_local_decl(expr const &e) const {
        return get_local_decl(fvar_name(e));
    }
```

```

    /* \brief Return type of the given free variable.
       \pre is_fvar(e) */
    expr get_type(expr const &e) const { return get_local_decl(e).get_type(); }

    /** Return the free variable associated with the given name.
        \pre get_local_decl(n) */
    expr get_local(name const &n) const;

    /** \brief Remove the given local decl. */
    void clear(local_decl const &d);

    expr mk_lambda(unsigned num, expr const *fvars, expr const &e,
                   bool remove_dead_let = false) const;
    expr mk_pi(unsigned num, expr const *fvars, expr const &e,
               bool remove_dead_let = false) const;
    expr mk_lambda(buffer<expr> const &fvars, expr const &e,
                   bool remove_dead_let = false) const {
        return mk_lambda(fvars.size(), fvars.data(), e, remove_dead_let);
    }
    expr mk_pi(buffer<expr> const &fvars, expr const &e,
               bool remove_dead_let = false) const {
        return mk_pi(fvars.size(), fvars.data(), e, remove_dead_let);
    }
    expr mk_lambda(expr const &fvar, expr const &e) {
        return mk_lambda(1, &fvar, e);
    }
    expr mk_pi(expr const &fvar, expr const &e) { return mk_pi(1, &fvar, e); }
    expr mk_lambda(std::initializer_list<expr> const &fvars, expr const &e) {
        return mk_lambda(fvars.size(), fvars.begin(), e);
    }
    expr mk_pi(std::initializer_list<expr> const &fvars, expr const &e) {
        return mk_pi(fvars.size(), fvars.begin(), e);
    }
};

void initialize_local_ctx();
void finalize_local_ctx();
}  // namespace lean
// ::::::::::::::
// quot.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura

Quotient types.
*/
#pragma once
#include "kernel/environment.h"

namespace lean {
class quot_consts {
    static name *g_quot;
    static name *g_quot_lift;
    static name *g_quot_ind;
    static name *g_quot_mk;

    friend bool quot_is_decl(name const &n);
    friend bool quot_is_rec(name const &n);
    template <typename WHNF>
    friend optional<expr> quot_reduce_rec(expr const &e, WHNF const &whnf);
    template <typename WHNF, typename IS_STUCK>
    friend optional<expr> quot_is_stuck(expr const &e, WHNF const &whnf,
                                        IS_STUCK const &is_stuck);
    friend class environment;
    friend void initialize_quot();
    friend void finalize_quot();
```

```
};

inline bool quot_is_decl(name const &n) {
    return n == *quot_consts::g_quot || n == *quot_consts::g_quot_lift ||
           n == *quot_consts::g_quot_ind || n == *quot_consts::g_quot_mk;
}

inline bool quot_is_rec(name const &n) {
    return n == *quot_consts::g_quot_lift || n == *quot_consts::g_quot_ind;
}

/** \brief Try to reduce a `quot` recursor application (i.e., `quot.lift` or
   `quot.ind` application).

    `whnf : expr -> expr` */
template <typename WHNF>
optional<expr> quot_reduce_rec(expr const &e, WHNF const &whnf) {
    expr const &fn = get_app_fn(e);
    if (!is_constant(fn)) return none_expr();
    unsigned mk_pos;
    unsigned arg_pos;
    if (const_name(fn) == *quot_consts::g_quot_lift) {
        mk_pos = 5;
        arg_pos = 3;
    } else if (const_name(fn) == *quot_consts::g_quot_ind) {
        mk_pos = 4;
        arg_pos = 3;
    } else {
        return none_expr();
    }
    buffer<expr> args;
    get_app_args(e, args);
    if (args.size() <= mk_pos) return none_expr();

    expr mk = whnf(args[mk_pos]);
    expr const &mk_fn = get_app_fn(mk);
    if (!is_constant(mk_fn) || const_name(mk_fn) != *quot_consts::g_quot_mk)
        return none_expr();

    expr const &f = args[arg_pos];
    expr r = mk_app(f, app_arg(mk));
    unsigned elim_arity = mk_pos + 1;
    if (args.size() > elim_arity)
        r = mk_app(r, args.size() - elim_arity, args.begin() + elim_arity);
    return some_expr(r);
}

/** \brief Return a non-none expression that is preventing the `quot` recursor
   application from being reduced.

    `whnf : expr -> expr`
    `is_stuck : expr -> optional<expr> */
template <typename WHNF, typename IS_STUCK>
optional<expr> quot_is_stuck(expr const &e, WHNF const &whnf,
                             IS_STUCK const &is_stuck) {
    expr const &fn = get_app_fn(e);
    if (!is_constant(fn)) return none_expr();
    unsigned mk_pos;
    if (const_name(fn) == *quot_consts::g_quot_lift) {
        mk_pos = 5;
    } else if (const_name(fn) == *quot_consts::g_quot_ind) {
        mk_pos = 4;
    } else {
        return none_expr();
    }

    buffer<expr> args;
    get_app_args(e, args);
    if (args.size() <= mk_pos) return none_expr();

```

```
    return is_stuck(whnf(args[mk_pos]));
}

void initialize_quot();
void finalize_quot();
}  // namespace lean
// ::::::::::::::
// replace_fn.h
// ::::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <lean/interrupt.h>

#include <tuple>

#include "kernel/expr.h"
#include "kernel/expr_maps.h"
#include "util/buffer.h"

namespace lean {
/**
   \brief Apply <tt>f</tt> to the subexpressions of a given expression.

   f is invoked for each subexpression \c s of the input expression e.
   In a call <tt>f(s, n)</tt>, n is the scope level, i.e., the number of
   bindings operators that enclosing \c s. The replaces only visits children of
   \c e if f return none_expr.
*/
expr replace(expr const &e,
             std::function<optional<expr>(expr const &, unsigned)> const &f,
             bool use_cache = true);
inline expr replace(expr const &e,
                    std::function<optional<expr>(expr const &)> const &f,
                    bool use_cache = true) {
    return replace(
        e, [&](expr const &e, unsigned) { return f(e); }, use_cache);
}
}  // namespace lean
// ::::::::::::::
// type_checker.h
// ::::::::::::::
/*
Copyright (c) 2013-14 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include <algorithm>
#include <memory>
#include <unordered_set>
#include <utility>

#include "kernel/environment.h"
#include "kernel/equiv_manager.h"
#include "kernel/expr_maps.h"
#include "kernel/local_ctx.h"
#include "util/lbool.h"
#include "util/name_generator.h"
#include "util/name_set.h"

namespace lean {
/** \brief Lean Type Checker. It can also be used to infer types, check whether
   a type \c A is convertible to a type \c B, etc. */
class type_checker {
```

```
public:
 class state {
     typedef expr_map<expr> infer_cache;
     typedef std::unordered_set<expr_pair, expr_pair_hash, expr_pair_eq>
         expr_pair_set;
     environment m_env;
     name_generator m_ngen;
     infer_cache m_infer_type[2];
     expr_map<expr> m_whnf_core;
     expr_map<expr> m_whnf;
     equiv_manager m_eqv_manager;
     expr_pair_set m_failure;
     friend type_checker;

    public:
     state(environment const &env);
     environment &env() { return m_env; }
     environment const &env() const { return m_env; }
     name_generator &ngen() { return m_ngen; }
 };

private:
 bool m_st_owner;
 state *m_st;
 local_ctx m_lctx;
 bool m_safe_only;
 /* When `m_lparams != nullptr, the `check` method makes sure all level
    parameters are in `m_lparams`. */
 names const *m_lparams;

 expr ensure_sort_core(expr e, expr const &s);
 expr ensure_pi_core(expr e, expr const &s);
 void check_level(level const &l);
 expr infer_fvar(expr const &e);
 expr infer_constant(expr const &e, bool infer_only);
 expr infer_lambda(expr const &e, bool infer_only);
 expr infer_pi(expr const &e, bool infer_only);
 expr infer_app(expr const &e, bool infer_only);
 expr infer_proj(expr const &e, bool infer_only);
 expr infer_let(expr const &e, bool infer_only);
 expr infer_type_core(expr const &e, bool infer_only);
 expr infer_type(expr const &e);

 enum class reduction_status { Continue, DefUnknown, DefEqual, DefDiff };
 optional<expr> reduce_recursor(expr const &e, bool cheap);
 optional<expr> reduce_proj(expr const &e, bool cheap);
 expr whnf_fvar(expr const &e, bool cheap);
 optional<constant_info> is_delta(expr const &e) const;
 optional<expr> unfold_definition_core(expr const &e);

 bool is_def_eq_binding(expr t, expr s);
 bool is_def_eq(level const &l1, level const &l2);
 bool is_def_eq(levels const &ls1, levels const &ls2);
 lbool quick_is_def_eq(expr const &t, expr const &s, bool use_hash = false);
 lbool is_def_eq_offset(expr const &t, expr const &s);
 bool is_def_eq_args(expr t, expr s);
 bool try_eta_expansion_core(expr const &t, expr const &s);
 bool try_eta_expansion(expr const &t, expr const &s) {
     return try_eta_expansion_core(t, s) || try_eta_expansion_core(s, t);
 }
 lbool try_string_lit_expansion_core(expr const &t, expr const &s);
 lbool try_string_lit_expansion(expr const &t, expr const &s);
 bool is_def_eq_app(expr const &t, expr const &s);
 bool is_def_eq_proof_irrel(expr const &t, expr const &s);
 bool failed_before(expr const &t, expr const &s) const;
 void cache_failure(expr const &t, expr const &s);
 reduction_status lazy_delta_reduction_step(expr &t_n, expr &s_n);
 lbool lazy_delta_reduction(expr &t_n, expr &s_n);
 bool is_def_eq_core(expr const &t, expr const &s);
 /** \brief Like \c check, but ignores undefined universes */
```

```
    expr check_ignore_undefined_universes(expr const &e);

    template <typename F>
    optional<expr> reduce_bin_nat_op(F const &f, expr const &e);
    template <typename F>
    optional<expr> reduce_bin_nat_pred(F const &f, expr const &e);
    optional<expr> reduce_nat(expr const &e);

  public:
    type_checker(state &st, local_ctx const &lctx, bool safe_only = true);
    type_checker(state &st, bool safe_only = true)
        : type_checker(st, local_ctx(), safe_only) {}
    type_checker(environment const &env, local_ctx const &lctx,
                 bool safe_only = true);
    type_checker(environment const &env, bool safe_only = true)
        : type_checker(env, local_ctx(), safe_only) {}
    type_checker(type_checker &&);
    type_checker(type_checker const &) = delete;
    ~type_checker();

    environment const &env() const { return m_st->m_env; }

    /** \brief Return the type of \c t.
        It does not check whether the input expression is type correct or not.
        The contract is: IF the input expression is type correct, then the
       inferred type is correct. Throw an exception if a type error is found. */
    expr infer(expr const &t) { return infer_type(t); }

    /** \brief Type check the given expression, and return the type of \c t.
        Throw an exception if a type error is found.  */
    expr check(expr const &t, names const &ps);
    /** \brief Like \c check, but ignores undefined universes */
    expr check(expr const &t) { return check_ignore_undefined_universes(t); }

    /** \brief Return true iff t is definitionally equal to s. */
    bool is_def_eq(expr const &t, expr const &s);
    /** \brief Return true iff t is a proposition. */
    bool is_prop(expr const &t);
    /** \brief Return the weak head normal form of \c t. */
    expr whnf(expr const &t);
    /** \brief Return a Pi if \c t is convertible to a Pi type. Throw an
       exception otherwise. The argument \c s is used when reporting errors */
    expr ensure_pi(expr const &t, expr const &s);
    expr ensure_pi(expr const &t) { return ensure_pi(t, t); }
    /** \brief Mare sure type of \c e is a Pi, and return it. Throw an exception
     * otherwise. */
    expr ensure_fun(expr const &e) { return ensure_pi(infer(e), e); }
    /** \brief Return a Sort if \c t is convertible to Sort. Throw an exception
       otherwise. The argument \c s is used when reporting errors. */
    expr ensure_sort(expr const &t, expr const &s);
    /** \brief Return a Sort if \c t is convertible to Sort. Throw an exception
     * otherwise. */
    expr ensure_sort(expr const &t) { return ensure_sort(t, t); }
    /** \brief Mare sure type of \c e is a sort, and return it. Throw an
     * exception otherwise. */
    expr ensure_type(expr const &e) { return ensure_sort(infer(e), e); }
    expr eta_expand(expr const &e);

    expr whnf_core(expr const &e, bool cheap = false);
    optional<expr> unfold_definition(expr const &e);
};

void initialize_type_checker();
void finalize_type_checker();
}  // namespace lean
// ::::::::::::::
// abstract.cpp
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <algorithm>
#include <utility>
#include <vector>

#include "kernel/abstract.h"
#include "kernel/replace_fn.h"

namespace lean {
expr abstract(expr const &e, unsigned n, expr const *subst) {
    lean_assert(std::all_of(subst, subst + n, [](expr const &e) {
        return !has_loose_bvars(e) && is_fvar(e);
    }));
    if (!has_fvar(e)) return e;
    return replace(e, [=](expr const &m, unsigned offset) -> optional<expr> {
        if (!has_fvar(m))
            return some_expr(
                m);  // expression m does not contain free variables
        if (is_fvar(m)) {
            unsigned i = n;
            while (i > 0) {
                --i;
                if (fvar_name(subst[i]) == fvar_name(m))
                    return some_expr(mk_bvar(offset + n - i - 1));
            }
            return none_expr();
        }
        return none_expr();
    });
}

expr abstract(expr const &e, name const &n) {
    expr fvar = mk_fvar(n);
    return abstract(e, 1, &fvar);
}

static object *lean_expr_abstract_core(object *e0, size_t n, object *subst) {
    lean_assert(n <= lean_array_size(subst));
    expr const &e = reinterpret_cast<expr const &>(e0);
    if (!has_fvar(e)) {
        lean_inc(e0);
        return e0;
    }
    expr r = replace(e, [=](expr const &m, unsigned offset) -> optional<expr> {
        if (!has_fvar(m))
            return some_expr(
                m);  // expression m does not contain free variables
        if (is_fvar(m)) {
            size_t i = n;
            while (i > 0) {
                --i;
                object *v = lean_array_get_core(subst, i);
                if (is_fvar_core(v) && fvar_name_core(v) == fvar_name(m))
                    return some_expr(mk_bvar(offset + n - i - 1));
            }
            return none_expr();
        }
        return none_expr();
    });
    return r.steal();
}

extern "C" object *lean_expr_abstract_range(object *e, object *n,
                                            object *subst) {
    if (!lean_is_scalar(n))
        return lean_expr_abstract_core(e, lean_array_size(subst), subst);
    else
```

```
        return lean_expr_abstract_core(
            e, std::min(lean_unbox(n), lean_array_size(subst)), subst);
}

extern "C" object *lean_expr_abstract(object *e, object *subst) {
    return lean_expr_abstract_core(e, lean_array_size(subst), subst);
}
}  // namespace lean
// ::::::::::::::
// declaration.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include "kernel/declaration.h"
#include "kernel/environment.h"
#include "kernel/for_each_fn.h"

namespace lean {

extern "C" object *lean_mk_reducibility_hints_regular(uint32 h);
extern "C" uint32 lean_reducibility_hints_get_height(object *o);

reducibility_hints reducibility_hints::mk_regular(unsigned h) {
    return reducibility_hints(lean_mk_reducibility_hints_regular(h));
}

unsigned reducibility_hints::get_height() const {
    return lean_reducibility_hints_get_height(to_obj_arg());
}

int compare(reducibility_hints const &h1, reducibility_hints const &h2) {
    if (h1.kind() == h2.kind()) {
        if (h1.kind() == reducibility_hints_kind::Regular) {
            if (h1.get_height() == h2.get_height())
                return 0; /* unfold both */
            else if (h1.get_height() > h2.get_height())
                return -1; /* unfold f1 */
            else
                return 1; /* unfold f2 */
            return h1.get_height() > h2.get_height() ? -1 : 1;
        } else {
            return 0; /* reduce both */
        }
    } else {
        if (h1.kind() == reducibility_hints_kind::Opaque) {
            return 1; /* reduce f2 */
        } else if (h2.kind() == reducibility_hints_kind::Opaque) {
            return -1; /* reduce f1 */
        } else if (h1.kind() == reducibility_hints_kind::Abbreviation) {
            return -1; /* reduce f1 */
        } else if (h2.kind() == reducibility_hints_kind::Abbreviation) {
            return 1; /* reduce f2 */
        } else {
            lean_unreachable();
        }
    }
}

constant_val::constant_val(name const &n, names const &lparams,
                           expr const &type)
    : object_ref(mk_cnstr(0, n, lparams, type)) {}

extern "C" object *lean_mk_axiom_val(object *n, object *lparams, object *type,
                                     uint8 is_unsafe);
extern "C" uint8 lean_axiom_val_is_unsafe(object *v);

```

```
axiom_val::axiom_val(name const &n, names const &lparams, expr const &type,
                     bool is_unsafe)
    : object_ref(lean_mk_axiom_val(n.to_obj_arg(), lparams.to_obj_arg(),
                                   type.to_obj_arg(), is_unsafe)) {}

bool axiom_val::is_unsafe() const {
    return lean_axiom_val_is_unsafe(to_obj_arg());
}

extern "C" object *lean_mk_definition_val(object *n, object *lparams,
                                          object *type, object *value,
                                          object *hints, uint8 safety);
extern "C" uint8 lean_definition_val_get_safety(object *v);

definition_val::definition_val(name const &n, names const &lparams,
                               expr const &type, expr const &val,
                               reducibility_hints const &hints,
                               definition_safety safety)
    : object_ref(lean_mk_definition_val(
          n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
          val.to_obj_arg(), hints.to_obj_arg(), static_cast<uint8>(safety))) {}

definition_safety definition_val::get_safety() const {
    return static_cast<definition_safety>(
        lean_definition_val_get_safety(to_obj_arg()));
}

theorem_val::theorem_val(name const &n, names const &lparams, expr const &type,
                         expr const &val)
    : object_ref(mk_cnstr(0, constant_val(n, lparams, type), val)) {}

extern "C" object *lean_mk_opaque_val(object *n, object *lparams, object *type,
                                      object *value, uint8 is_unsafe);
extern "C" uint8 lean_opaque_val_is_unsafe(object *v);

opaque_val::opaque_val(name const &n, names const &lparams, expr const &type,
                       expr const &val, bool is_unsafe)
    : object_ref(lean_mk_opaque_val(n.to_obj_arg(), lparams.to_obj_arg(),
                                    type.to_obj_arg(), val.to_obj_arg(),
                                    is_unsafe)) {}

bool opaque_val::is_unsafe() const {
    return lean_opaque_val_is_unsafe(to_obj_arg());
}

extern "C" object *lean_mk_quot_val(object *n, object *lparams, object *type,
                                    uint8 k);
extern "C" uint8 lean_quot_val_kind(object *v);

quot_val::quot_val(name const &n, names const &lparams, expr const &type,
                   quot_kind k)
    : object_ref(lean_mk_quot_val(n.to_obj_arg(), lparams.to_obj_arg(),
                                  type.to_obj_arg(), static_cast<uint8>(k))) {}

quot_kind quot_val::get_quot_kind() const {
    return static_cast<quot_kind>(lean_quot_val_kind(to_obj_arg()));
}

recursor_rule::recursor_rule(name const &cnstr, unsigned nfields,
                             expr const &rhs)
    : object_ref(mk_cnstr(0, cnstr, nat(nfields), rhs)) {}

extern "C" object *lean_mk_inductive_val(object *n, object *lparams,
                                         object *type, object *nparams,
                                         object *nindices, object *all,
                                         object *cnstrs, uint8 rec,
                                         uint8 unsafe, uint8 is_refl,
                                         uint8 is_nested);
extern "C" uint8 lean_inductive_val_is_rec(object *v);
extern "C" uint8 lean_inductive_val_is_unsafe(object *v);
```

```
extern "C" uint8 lean_inductive_val_is_reflexive(object *v);
extern "C" uint8 lean_inductive_val_is_nested(object *v);

inductive_val::inductive_val(name const &n, names const &lparams,
                             expr const &type, unsigned nparams,
                             unsigned nindices, names const &all,
                             names const &cnstrs, bool rec, bool unsafe,
                             bool is_refl, bool is_nested)
    : object_ref(lean_mk_inductive_val(
          n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
          nat(nparams).to_obj_arg(), nat(nindices).to_obj_arg(),
          all.to_obj_arg(), cnstrs.to_obj_arg(), rec, unsafe, is_refl,
          is_nested)) {}

bool inductive_val::is_rec() const {
    return lean_inductive_val_is_rec(to_obj_arg());
}
bool inductive_val::is_unsafe() const {
    return lean_inductive_val_is_unsafe(to_obj_arg());
}
bool inductive_val::is_reflexive() const {
    return lean_inductive_val_is_reflexive(to_obj_arg());
}
bool inductive_val::is_nested() const {
    return lean_inductive_val_is_nested(to_obj_arg());
}

extern "C" object *lean_mk_constructor_val(object *n, object *lparams,
                                           object *type, object *induct,
                                           object *cidx, object *nparams,
                                           object *nfields, uint8 unsafe);
extern "C" uint8 lean_constructor_val_is_unsafe(object *v);

constructor_val::constructor_val(name const &n, names const &lparams,
                                 expr const &type, name const &induct,
                                 unsigned cidx, unsigned nparams,
                                 unsigned nfields, bool is_unsafe)
    : object_ref(lean_mk_constructor_val(
          n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
          induct.to_obj_arg(), nat(cidx).to_obj_arg(),
          nat(nparams).to_obj_arg(), nat(nfields).to_obj_arg(), is_unsafe)) {}

bool constructor_val::is_unsafe() const {
    return lean_constructor_val_is_unsafe(to_obj_arg());
}

extern "C" object *lean_mk_recursor_val(object *n, object *lparams,
                                        object *type, object *all,
                                        object *nparams, object *nindices,
                                        object *nmotives, object *nminors,
                                        object *rules, uint8 k, uint8 unsafe);
extern "C" uint8 lean_recursor_k(object *v);
extern "C" uint8 lean_recursor_is_unsafe(object *v);

recursor_val::recursor_val(name const &n, names const &lparams,
                           expr const &type, names const &all, unsigned nparams,
                           unsigned nindices, unsigned nmotives,
                           unsigned nminors, recursor_rules const &rules,
                           bool k, bool is_unsafe)
    : object_ref(lean_mk_recursor_val(
          n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
          all.to_obj_arg(), nat(nparams).to_obj_arg(),
          nat(nindices).to_obj_arg(), nat(nmotives).to_obj_arg(),
          nat(nminors).to_obj_arg(), rules.to_obj_arg(), k, is_unsafe)) {}

bool recursor_val::is_k() const { return lean_recursor_k(to_obj_arg()); }
bool recursor_val::is_unsafe() const {
    return lean_recursor_is_unsafe(to_obj_arg());
}
```

```
bool declaration::is_unsafe() const {
    switch (kind()) {
        case declaration_kind::Definition:
            return to_definition_val().get_safety() ==
                   definition_safety::unsafe;
        case declaration_kind::Axiom:
            return to_axiom_val().is_unsafe();
        case declaration_kind::Theorem:
            return false;
        case declaration_kind::Opaque:
            return to_opaque_val().is_unsafe();
        case declaration_kind::Inductive:
            return inductive_decl(*this).is_unsafe();
        case declaration_kind::Quot:
            return false;
        case declaration_kind::MutualDefinition:
            return true;
    }
    lean_unreachable();
}

bool use_unsafe(environment const &env, expr const &e) {
    bool found = false;
    for_each(e, [&](expr const &e, unsigned) {
        if (found) return false;
        if (is_constant(e)) {
            if (auto info = env.find(const_name(e))) {
                if (info->is_unsafe()) {
                    found = true;
                    return false;
                }
            }
        }
        return true;
    });
    return found;
}

static declaration *g_dummy = nullptr;
declaration::declaration() : declaration(*g_dummy) {}

static unsigned get_max_height(environment const &env, expr const &v) {
    unsigned h = 0;
    for_each(v, [&](expr const &e, unsigned) {
        if (is_constant(e)) {
            auto d = env.find(const_name(e));
            if (d && d->get_hints().get_height() > h)
                h = d->get_hints().get_height();
        }
        return true;
    });
    return h;
}

definition_val mk_definition_val(environment const &env, name const &n,
                                 names const &params, expr const &t,
                                 expr const &v, definition_safety s) {
    unsigned h = get_max_height(env, v);
    return definition_val(n, params, t, v,
                          reducibility_hints::mk_regular(h + 1), s);
}

declaration mk_definition(name const &n, names const &params, expr const &t,
                          expr const &v, reducibility_hints const &h,
                          definition_safety safety) {
    return declaration(
        mk_cnstr(static_cast<unsigned>(declaration_kind::Definition),
                 definition_val(n, params, t, v, h, safety)));
}
```

```
declaration mk_definition(environment const &env, name const &n,
                          names const &params, expr const &t, expr const &v,
                          definition_safety safety) {
    return declaration(
        mk_cnstr(static_cast<unsigned>(declaration_kind::Definition),
                 mk_definition_val(env, n, params, t, v, safety)));
}

declaration mk_opaque(name const &n, names const &params, expr const &t,
                      expr const &v, bool is_unsafe) {
    return declaration(mk_cnstr(static_cast<unsigned>(declaration_kind::Opaque),
                                opaque_val(n, params, t, v, is_unsafe)));
}

declaration mk_axiom(name const &n, names const &params, expr const &t,
                     bool unsafe) {
    return declaration(mk_cnstr(static_cast<unsigned>(declaration_kind::Axiom),
                                axiom_val(n, params, t, unsafe)));
}

static definition_safety to_safety(bool unsafe) {
    return unsafe ? definition_safety::unsafe : definition_safety::safe;
}

declaration mk_definition_inferring_unsafe(environment const &env,
                                           name const &n, names const &params,
                                           expr const &t, expr const &v,
                                           reducibility_hints const &hints) {
    bool unsafe = use_unsafe(env, t) || use_unsafe(env, v);
    return mk_definition(n, params, t, v, hints, to_safety(unsafe));
}

declaration mk_definition_inferring_unsafe(environment const &env,
                                           name const &n, names const &params,
                                           expr const &t, expr const &v) {
    bool unsafe = use_unsafe(env, t) && use_unsafe(env, v);
    unsigned h = get_max_height(env, v);
    return mk_definition(n, params, t, v, reducibility_hints::mk_regular(h + 1),
                         to_safety(unsafe));
}

inductive_type::inductive_type(name const &id, expr const &type,
                               constructors const &cnstrs)
    : object_ref(mk_cnstr(0, id, type, cnstrs)) {}

extern "C" object *lean_mk_inductive_decl(object *lparams, object *nparams,
                                          object *types, uint8 unsafe);
extern "C" uint8 lean_is_unsafe_inductive_decl(object *d);

declaration mk_inductive_decl(names const &lparams, nat const &nparams,
                              inductive_types const &types, bool is_unsafe) {
    return declaration(lean_mk_inductive_decl(lparams.to_obj_arg(),
                                              nparams.to_obj_arg(),
                                              types.to_obj_arg(), is_unsafe));
}

bool inductive_decl::is_unsafe() const {
    return lean_is_unsafe_inductive_decl(to_obj_arg());
}

// =========================================
// Constant info
constant_info::constant_info() : constant_info(*g_dummy) {}

constant_info::constant_info(declaration const &d) : object_ref(d.raw()) {
    lean_assert(d.is_definition() || d.is_theorem() || d.is_axiom() ||
                d.is_opaque());
    inc_ref(d.raw());
}

```

```cpp
constant_info::constant_info(definition_val const &v)
    : object_ref(
          mk_cnstr(static_cast<unsigned>(constant_info_kind::Definition), v)) {}

constant_info::constant_info(quot_val const &v)
    : object_ref(mk_cnstr(static_cast<unsigned>(constant_info_kind::Quot), v)) {
}

constant_info::constant_info(inductive_val const &v)
    : object_ref(
          mk_cnstr(static_cast<unsigned>(constant_info_kind::Inductive), v)) {}

constant_info::constant_info(constructor_val const &v)
    : object_ref(mk_cnstr(
          static_cast<unsigned>(constant_info_kind::Constructor), v)) {}

constant_info::constant_info(recursor_val const &v)
    : object_ref(
          mk_cnstr(static_cast<unsigned>(constant_info_kind::Recursor), v)) {}

static reducibility_hints *g_opaque = nullptr;

reducibility_hints const &constant_info::get_hints() const {
    if (is_definition())
        return static_cast<reducibility_hints const &>(
            cnstr_get_ref(to_val(), 2));
    else
        return *g_opaque;
}

bool constant_info::is_unsafe() const {
    switch (kind()) {
        case constant_info_kind::Axiom:
            return to_axiom_val().is_unsafe();
        case constant_info_kind::Definition:
            return to_definition_val().get_safety() ==
                   definition_safety::unsafe;
        case constant_info_kind::Theorem:
            return false;
        case constant_info_kind::Opaque:
            return to_opaque_val().is_unsafe();
        case constant_info_kind::Quot:
            return false;
        case constant_info_kind::Inductive:
            return to_inductive_val().is_unsafe();
        case constant_info_kind::Constructor:
            return to_constructor_val().is_unsafe();
        case constant_info_kind::Recursor:
            return to_recursor_val().is_unsafe();
    }
    lean_unreachable();
}

void initialize_declaration() {
    g_opaque = new reducibility_hints(reducibility_hints::mk_opaque());
    mark_persistent(g_opaque->raw());
    g_dummy = new declaration(mk_axiom(name(), names(), expr()));
    mark_persistent(g_dummy->raw());
}

void finalize_declaration() {
    delete g_dummy;
    delete g_opaque;
}
}  // namespace lean
// ::::::::::::
// environment.cpp
// ::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/sstream.h>
#include <lean/thread.h>

#include <limits>
#include <utility>
#include <vector>

#include "kernel/environment.h"
#include "kernel/kernel_exception.h"
#include "kernel/quot.h"
#include "kernel/type_checker.h"
#include "util/io.h"
#include "util/map_foreach.h"

namespace lean {
extern "C" object *lean_environment_add(object *, object *);
extern "C" object *lean_mk_empty_environment(uint32, object *);
extern "C" object *lean_environment_find(object *, object *);
extern "C" uint32 lean_environment_trust_level(object *);
extern "C" object *lean_environment_mark_quot_init(object *);
extern "C" uint8 lean_environment_quot_init(object *);
extern "C" object *lean_register_extension(object *);
extern "C" object *lean_get_extension(object *, object *);
extern "C" object *lean_set_extension(object *, object *, object *);
extern "C" object *lean_environment_set_main_module(object *, object *);
extern "C" object *lean_environment_main_module(object *);

environment mk_empty_environment(uint32 trust_lvl) {
    return get_io_result<environment>(
        lean_mk_empty_environment(trust_lvl, io_mk_world()));
}

environment::environment(unsigned trust_lvl)
    : object_ref(mk_empty_environment(trust_lvl)) {}

void environment::set_main_module(name const &n) {
    m_obj = lean_environment_set_main_module(m_obj, n.to_obj_arg());
}

name environment::get_main_module() const {
    return name(lean_environment_main_module(to_obj_arg()));
}

unsigned environment::trust_lvl() const {
    return lean_environment_trust_level(to_obj_arg());
}

bool environment::is_quot_initialized() const {
    return lean_environment_quot_init(to_obj_arg()) != 0;
}

void environment::mark_quot_initialized() {
    m_obj = lean_environment_mark_quot_init(m_obj);
}

optional<constant_info> environment::find(name const &n) const {
    return to_optional<constant_info>(
        lean_environment_find(to_obj_arg(), n.to_obj_arg()));
}

constant_info environment::get(name const &n) const {
    object *o = lean_environment_find(to_obj_arg(), n.to_obj_arg());
    if (is_scalar(o)) throw unknown_constant_exception(*this, n);
    constant_info r(cnstr_get(o, 0), true);
    dec(o);
    return r;
```

```
}

static void check_no_metavar(environment const &env, name const &n,
                             expr const &e) {
    if (has_metavar(e)) throw declaration_has_metavars_exception(env, n, e);
}

static void check_no_fvar(environment const &env, name const &n,
                          expr const &e) {
    if (has_fvar(e)) throw declaration_has_free_vars_exception(env, n, e);
}

void check_no_metavar_no_fvar(environment const &env, name const &n,
                              expr const &e) {
    check_no_metavar(env, n, e);
    check_no_fvar(env, n, e);
}

static void check_name(environment const &env, name const &n) {
    if (env.find(n)) throw already_declared_exception(env, n);
}

void environment::check_name(name const &n) const {
    ::lean::check_name(*this, n);
}

static void check_duplicated_univ_params(environment const &env, names ls) {
    while (!is_nil(ls)) {
        auto const &p = head(ls);
        ls = tail(ls);
        if (std::find(ls.begin(), ls.end(), p) != ls.end()) {
            throw kernel_exception(
                env, sstream() << "failed to add declaration to environment, "
                               << "duplicate universe level parameter: '" << p
                               << "'");
        }
    }
}

void environment::check_duplicated_univ_params(names ls) const {
    ::lean::check_duplicated_univ_params(*this, ls);
}

static void check_constant_val(environment const &env, constant_val const &v,
                               type_checker &checker) {
    check_name(env, v.get_name());
    check_duplicated_univ_params(env, v.get_lparams());
    check_no_metavar_no_fvar(env, v.get_name(), v.get_type());
    expr sort = checker.check(v.get_type(), v.get_lparams());
    checker.ensure_sort(sort, v.get_type());
}

static void check_constant_val(environment const &env, constant_val const &v,
                               bool safe_only) {
    type_checker checker(env, safe_only);
    check_constant_val(env, v, checker);
}

void environment::add_core(constant_info const &info) {
    m_obj = lean_environment_add(m_obj, info.to_obj_arg());
}

environment environment::add(constant_info const &info) const {
    return environment(lean_environment_add(to_obj_arg(), info.to_obj_arg()));
}

environment environment::add_axiom(declaration const &d, bool check) const {
    axiom_val const &v = d.to_axiom_val();
    if (check) check_constant_val(*this, v.to_constant_val(), !d.is_unsafe());
    return add(constant_info(d));
```

```cpp
}

environment environment::add_definition(declaration const &d,
                                        bool check) const {
    definition_val const &v = d.to_definition_val();
    if (v.is_unsafe()) {
        /* Meta definition can be recursive.
           So, we check the header, add, and then type check the body. */
        if (check) {
            bool safe_only = false;
            type_checker checker(*this, safe_only);
            check_constant_val(*this, v.to_constant_val(), checker);
        }
        environment new_env = add(constant_info(d));
        if (check) {
            bool safe_only = false;
            type_checker checker(new_env, safe_only);
            check_no_metavar_no_fvar(new_env, v.get_name(), v.get_value());
            expr val_type = checker.check(v.get_value(), v.get_lparams());
            if (!checker.is_def_eq(val_type, v.get_type()))
                throw definition_type_mismatch_exception(new_env, d, val_type);
        }
        return new_env;
    } else {
        if (check) {
            type_checker checker(*this);
            check_constant_val(*this, v.to_constant_val(), checker);
            check_no_metavar_no_fvar(*this, v.get_name(), v.get_value());
            expr val_type = checker.check(v.get_value(), v.get_lparams());
            if (!checker.is_def_eq(val_type, v.get_type()))
                throw definition_type_mismatch_exception(*this, d, val_type);
        }
        return add(constant_info(d));
    }
}

environment environment::add_theorem(declaration const &d, bool check) const {
    theorem_val const &v = d.to_theorem_val();
    if (check) {
        // TODO(Leo): we must add support for handling tasks here
        type_checker checker(*this);
        check_constant_val(*this, v.to_constant_val(), checker);
        check_no_metavar_no_fvar(*this, v.get_name(), v.get_value());
        expr val_type = checker.check(v.get_value(), v.get_lparams());
        if (!checker.is_def_eq(val_type, v.get_type()))
            throw definition_type_mismatch_exception(*this, d, val_type);
    }
    return add(constant_info(d));
}

environment environment::add_opaque(declaration const &d, bool check) const {
    opaque_val const &v = d.to_opaque_val();
    if (check) {
        type_checker checker(*this);
        check_constant_val(*this, v.to_constant_val(), checker);
        expr val_type = checker.check(v.get_value(), v.get_lparams());
        if (!checker.is_def_eq(val_type, v.get_type()))
            throw definition_type_mismatch_exception(*this, d, val_type);
    }
    return add(constant_info(d));
}

environment environment::add_mutual(declaration const &d, bool check) const {
    definition_vals const &vs = d.to_definition_vals();
    if (empty(vs))
        throw kernel_exception(*this, "invalid empty mutual definition");
    definition_safety safety = head(vs).get_safety();
    if (safety == definition_safety::safe)
        throw kernel_exception(*this,
                               "invalid mutual definition, declaration is not "
```

```
                               "tagged as unsafe/partial");
    bool safe_only = safety == definition_safety::partial;
    /* Check declarations header */
    if (check) {
        type_checker checker(*this, safe_only);
        for (definition_val const &v : vs) {
            if (v.get_safety() != safety)
                throw kernel_exception(
                    *this,
                    "invalid mutual definition, declarations must have the "
                    "same safety annotation");
            check_constant_val(*this, v.to_constant_val(), checker);
        }
    }
    /* Add declarations */
    environment new_env = *this;
    for (definition_val const &v : vs) {
        new_env.add_core(constant_info(v));
    }
    /* Check actual definitions */
    if (check) {
        type_checker checker(new_env, safe_only);
        for (definition_val const &v : vs) {
            check_no_metavar_no_fvar(new_env, v.get_name(), v.get_value());
            expr val_type = checker.check(v.get_value(), v.get_lparams());
            if (!checker.is_def_eq(val_type, v.get_type()))
                throw definition_type_mismatch_exception(new_env, d, val_type);
        }
    }
    return new_env;
}

environment environment::add(declaration const &d, bool check) const {
    switch (d.kind()) {
        case declaration_kind::Axiom:
            return add_axiom(d, check);
        case declaration_kind::Definition:
            return add_definition(d, check);
        case declaration_kind::Theorem:
            return add_theorem(d, check);
        case declaration_kind::Opaque:
            return add_opaque(d, check);
        case declaration_kind::MutualDefinition:
            return add_mutual(d, check);
        case declaration_kind::Quot:
            return add_quot();
        case declaration_kind::Inductive:
            return add_inductive(d);
    }
    lean_unreachable();
}

extern "C" object *lean_add_decl(object *env, object *decl) {
    return catch_kernel_exceptions<environment>(
        [&]() { return environment(env).add(declaration(decl, true)); });
}

void environment::for_each_constant(
    std::function<void(constant_info const &d)> const &f) const {
    smap_foreach(cnstr_get(raw(), 1), [&](object *, object *v) {
        constant_info cinfo(v, true);
        f(cinfo);
    });
}

extern "C" obj_res lean_display_stats(obj_arg env, obj_arg w);

void environment::display_stats() const {
    dec_ref(lean_display_stats(to_obj_arg(), io_mk_world()));
}
```

```cpp

void initialize_environment() {}

void finalize_environment() {}
}  // namespace lean
// ::::::::::::::
// equiv_manager.cpp
// ::::::::::::::
/*
Copyright (c) 2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/flet.h>
#include <lean/interrupt.h>

#include "kernel/equiv_manager.h"

namespace lean {
auto equiv_manager::mk_node() -> node_ref {
    node_ref r = m_nodes.size();
    node n;
    n.m_parent = r;
    n.m_rank = 0;
    m_nodes.push_back(n);
    return r;
}

auto equiv_manager::find(node_ref n) -> node_ref {
    while (true) {
        node_ref p = m_nodes[n].m_parent;
        if (p == n) return p;
        n = p;
    }
}

void equiv_manager::merge(node_ref n1, node_ref n2) {
    node_ref r1 = find(n1);
    node_ref r2 = find(n2);
    if (r1 != r2) {
        node &ref1 = m_nodes[r1];
        node &ref2 = m_nodes[r2];
        if (ref1.m_rank < ref2.m_rank) {
            ref1.m_parent = r2;
        } else if (ref1.m_rank > ref2.m_rank) {
            ref2.m_parent = r1;
        } else {
            ref2.m_parent = r1;
            ref1.m_rank++;
        }
    }
}

auto equiv_manager::to_node(expr const &e) -> node_ref {
    auto it = m_to_node.find(e);
    if (it != m_to_node.end()) return it->second;
    node_ref r = mk_node();
    m_to_node.insert(mk_pair(e, r));
    return r;
}

bool equiv_manager::is_equiv_core(expr const &a, expr const &b) {
    if (is_eqp(a, b)) return true;
    if (m_use_hash && hash(a) != hash(b)) return false;
    if (is_bvar(a) && is_bvar(b)) return bvar_idx(a) == bvar_idx(b);
    node_ref r1 = find(to_node(a));
    node_ref r2 = find(to_node(b));
    if (r1 == r2) return true;
    // fall back to structural equality
```

```
    if (a.kind() != b.kind()) return false;
    check_system("expression equivalence test");
    bool result = false;
    switch (a.kind()) {
        case expr_kind::BVar:
            lean_unreachable();  // LCOV_EXCL_LINE
        case expr_kind::Const:
            result = const_name(a) == const_name(b) &&
                     compare(const_levels(a), const_levels(b),
                             [](level const &l1, level const &l2) {
                                 return l1 == l2;
                             });
            break;
        case expr_kind::MVar:
            result = mvar_name(a) == mvar_name(b);
            break;
        case expr_kind::FVar:
            result = fvar_name(a) == fvar_name(b);
            break;
        case expr_kind::App:
            result = is_equiv_core(app_fn(a), app_fn(b)) &&
                     is_equiv_core(app_arg(a), app_arg(b));
            break;
        case expr_kind::Lambda:
        case expr_kind::Pi:
            result = is_equiv_core(binding_domain(a), binding_domain(b)) &&
                     is_equiv_core(binding_body(a), binding_body(b));
            break;
        case expr_kind::Sort:
            result = sort_level(a) == sort_level(b);
            break;
        case expr_kind::Lit:
            result = lit_value(a) == lit_value(b);
            break;
        case expr_kind::MData:
            result = is_equiv_core(mdata_expr(a), mdata_expr(b));
            break;
        case expr_kind::Proj:
            result = is_equiv_core(proj_expr(a), proj_expr(b)) &&
                     proj_idx(a) == proj_idx(b);
            break;
        case expr_kind::Let:
            result = is_equiv_core(let_type(a), let_type(b)) &&
                     is_equiv_core(let_value(a), let_value(b)) &&
                     is_equiv_core(let_body(a), let_body(b));
            break;
    }
    if (result) merge(r1, r2);
    return result;
}

bool equiv_manager::is_equiv(expr const &a, expr const &b, bool use_hash) {
    flet<bool> set(m_use_hash, use_hash);
    return is_equiv_core(a, b);
}

void equiv_manager::add_equiv(expr const &e1, expr const &e2) {
    node_ref r1 = to_node(e1);
    node_ref r2 = to_node(e2);
    merge(r1, r2);
}
}  // namespace lean
// ::::::::::::::
// expr_cache.cpp
// ::::::::::::::
/*
Copyright (c) 2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
```

```
*/
#include "kernel/expr_cache.h"

namespace lean {
expr *expr_cache::find(expr const &e) {
    unsigned i = hash(e) % m_capacity;
    if (m_cache[i].m_expr && is_bi_equal(*m_cache[i].m_expr, e))
        return &m_cache[i].m_result;
    else
        return nullptr;
}

void expr_cache::insert(expr const &e, expr const &v) {
    unsigned i = hash(e) % m_capacity;
    if (!m_cache[i].m_expr) m_used.push_back(i);
    m_cache[i].m_expr = e;
    m_cache[i].m_result = v;
}

void expr_cache::clear() {
    for (unsigned i : m_used) {
        m_cache[i].m_expr = none_expr();
        m_cache[i].m_result = expr();
    }
    m_used.clear();
}
}  // namespace lean
// ::::::::::::::
// expr.cpp
// ::::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
        Soonho Kong
*/
#include <lean/hash.h>

#include <algorithm>
#include <limits>
#include <sstream>
#include <string>
#include <vector>

#include "kernel/abstract.h"
#include "kernel/expr.h"
#include "kernel/expr_eq_fn.h"
#include "kernel/expr_sets.h"
#include "kernel/for_each_fn.h"
#include "kernel/instantiate.h"
#include "kernel/replace_fn.h"
#include "util/buffer.h"
#include "util/list_fn.h"

namespace lean {
/* Expression literal values */
literal::literal(char const *v)
    : object_ref(mk_cnstr(static_cast<unsigned>(literal_kind::String),
                          mk_string(v))) {}

literal::literal(unsigned v)
    : object_ref(
          mk_cnstr(static_cast<unsigned>(literal_kind::Nat), mk_nat_obj(v))) {}

literal::literal(mpz const &v)
    : object_ref(
          mk_cnstr(static_cast<unsigned>(literal_kind::Nat), mk_nat_obj(v))) {}

literal::literal(nat const &v)
```

```
    : object_ref(mk_cnstr(static_cast<unsigned>(literal_kind::Nat), v)) {}

bool operator==(literal const &a, literal const &b) {
    if (a.kind() != b.kind()) return false;
    switch (a.kind()) {
        case literal_kind::String:
            return a.get_string() == b.get_string();
        case literal_kind::Nat:
            return a.get_nat() == b.get_nat();
    }
    lean_unreachable();
}

bool operator<(literal const &a, literal const &b) {
    if (a.kind() != b.kind())
        return static_cast<unsigned>(a.kind()) <
               static_cast<unsigned>(b.kind());
    switch (a.kind()) {
        case literal_kind::String:
            return a.get_string() < b.get_string();
        case literal_kind::Nat:
            return a.get_nat() < b.get_nat();
    }
    lean_unreachable();
}

bool is_atomic(expr const &e) {
    switch (e.kind()) {
        case expr_kind::Const:
        case expr_kind::Sort:
        case expr_kind::BVar:
        case expr_kind::Lit:
        case expr_kind::MVar:
        case expr_kind::FVar:
            return true;
        case expr_kind::App:
        case expr_kind::Lambda:
        case expr_kind::Pi:
        case expr_kind::Let:
        case expr_kind::MData:
        case expr_kind::Proj:
            return false;
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

extern "C" uint8 lean_expr_binder_info(object *e);
binder_info binding_info(expr const &e) {
    return static_cast<binder_info>(lean_expr_binder_info(e.to_obj_arg()));
}

extern "C" object *lean_lit_type(obj_arg e);
expr lit_type(literal const &lit) {
    return expr(lean_lit_type(lit.to_obj_arg()));
}

extern "C" usize lean_expr_hash(obj_arg e);
unsigned hash(expr const &e) { return lean_expr_hash(e.to_obj_arg()); }

extern "C" uint8 lean_expr_has_fvar(obj_arg e);
bool has_fvar(expr const &e) { return lean_expr_has_fvar(e.to_obj_arg()); }

extern "C" uint8 lean_expr_has_expr_mvar(obj_arg e);
bool has_expr_mvar(expr const &e) {
    return lean_expr_has_expr_mvar(e.to_obj_arg());
}

extern "C" uint8 lean_expr_has_level_mvar(obj_arg e);
bool has_univ_mvar(expr const &e) {
    return lean_expr_has_level_mvar(e.to_obj_arg());
```

```
}

extern "C" uint8 lean_expr_has_level_param(obj_arg e);
bool has_univ_param(expr const &e) {
    return lean_expr_has_level_param(e.to_obj_arg());
}

extern "C" unsigned lean_expr_loose_bvar_range(object *e);
unsigned get_loose_bvar_range(expr const &e) {
    return lean_expr_loose_bvar_range(e.to_obj_arg());
}

// =======================================
// Constructors

static expr *g_dummy = nullptr;

static expr const &get_dummy() {
    if (!g_dummy) {
        g_dummy = new expr(mk_constant("__expr_for_default_constructor__"));
        mark_persistent(g_dummy->raw());
    }
    return *g_dummy;
}

expr::expr() : expr(get_dummy()) {}

extern "C" object *lean_expr_mk_lit(obj_arg l);
expr mk_lit(literal const &l) { return expr(lean_expr_mk_lit(l.to_obj_arg())); }

extern "C" object *lean_expr_mk_mdata(obj_arg m, obj_arg e);
expr mk_mdata(kvmap const &m, expr const &e) {
    return expr(lean_expr_mk_mdata(m.to_obj_arg(), e.to_obj_arg()));
}

extern "C" object *lean_expr_mk_proj(obj_arg s, obj_arg idx, obj_arg e);
expr mk_proj(name const &s, nat const &idx, expr const &e) {
    return expr(
        lean_expr_mk_proj(s.to_obj_arg(), idx.to_obj_arg(), e.to_obj_arg()));
}

extern "C" object *lean_expr_mk_bvar(obj_arg idx);
expr mk_bvar(nat const &idx) {
    return expr(lean_expr_mk_bvar(idx.to_obj_arg()));
}

extern "C" object *lean_expr_mk_fvar(obj_arg n);
expr mk_fvar(name const &n) { return expr(lean_expr_mk_fvar(n.to_obj_arg())); }

extern "C" object *lean_expr_mk_mvar(object *n);
expr mk_mvar(name const &n) { return expr(lean_expr_mk_mvar(n.to_obj_arg())); }

extern "C" object *lean_expr_mk_const(obj_arg n, obj_arg ls);
expr mk_const(name const &n, levels const &ls) {
    return expr(lean_expr_mk_const(n.to_obj_arg(), ls.to_obj_arg()));
}

extern "C" object *lean_expr_mk_app(obj_arg f, obj_arg a);
expr mk_app(expr const &f, expr const &a) {
    return expr(lean_expr_mk_app(f.to_obj_arg(), a.to_obj_arg()));
}

extern "C" object *lean_expr_mk_sort(obj_arg l);
expr mk_sort(level const &l) { return expr(lean_expr_mk_sort(l.to_obj_arg())); }

extern "C" object *lean_expr_mk_lambda(obj_arg n, obj_arg t, obj_arg e,
                                       uint8 bi);
expr mk_lambda(name const &n, expr const &t, expr const &e, binder_info bi) {
    return expr(lean_expr_mk_lambda(n.to_obj_arg(), t.to_obj_arg(),
                                    e.to_obj_arg(), static_cast<uint8>(bi)));
```

```
}

extern "C" object *lean_expr_mk_forall(obj_arg n, obj_arg t, obj_arg e,
                                        uint8 bi);
expr mk_pi(name const &n, expr const &t, expr const &e, binder_info bi) {
    return expr(lean_expr_mk_forall(n.to_obj_arg(), t.to_obj_arg(),
                                    e.to_obj_arg(), static_cast<uint8>(bi)));
}

static name *g_default_name = nullptr;
expr mk_arrow(expr const &t, expr const &e) {
    return mk_pi(*g_default_name, t, e, mk_binder_info());
}

extern "C" object *lean_expr_mk_let(object *n, object *t, object *v, object *b);
expr mk_let(name const &n, expr const &t, expr const &v, expr const &b) {
    return expr(lean_expr_mk_let(n.to_obj_arg(), t.to_obj_arg(), v.to_obj_arg(),
                                 b.to_obj_arg()));
}

static expr *g_Prop = nullptr;
static expr *g_Type0 = nullptr;
expr mk_Prop() { return *g_Prop; }
expr mk_Type() { return *g_Type0; }

// =======================================
// Auxiliary constructors and accessors

expr mk_app(expr const &f, unsigned num_args, expr const *args) {
    expr r = f;
    for (unsigned i = 0; i < num_args; i++) r = mk_app(r, args[i]);
    return r;
}

expr mk_app(unsigned num_args, expr const *args) {
    lean_assert(num_args >= 2);
    return mk_app(mk_app(args[0], args[1]), num_args - 2, args + 2);
}

expr mk_app(expr const &f, list<expr> const &args) {
    buffer<expr> _args;
    to_buffer(args, _args);
    return mk_app(f, _args);
}

expr mk_rev_app(expr const &f, unsigned num_args, expr const *args) {
    expr r = f;
    unsigned i = num_args;
    while (i > 0) {
        --i;
        r = mk_app(r, args[i]);
    }
    return r;
}

expr mk_rev_app(unsigned num_args, expr const *args) {
    lean_assert(num_args >= 2);
    return mk_rev_app(mk_app(args[num_args - 1], args[num_args - 2]),
                      num_args - 2, args);
}

expr const &get_app_args(expr const &e, buffer<expr> &args) {
    unsigned sz = args.size();
    expr const *it = &e;
    while (is_app(*it)) {
        args.push_back(app_arg(*it));
        it = &(app_fn(*it));
    }
    std::reverse(args.begin() + sz, args.end());
    return *it;
```

```
}

expr const &get_app_args_at_most(expr const &e, unsigned num,
                                 buffer<expr> &args) {
    unsigned sz = args.size();
    expr const *it = &e;
    unsigned i = 0;
    while (is_app(*it)) {
        if (i == num) break;
        args.push_back(app_arg(*it));
        it = &(app_fn(*it));
        i++;
    }
    std::reverse(args.begin() + sz, args.end());
    return *it;
}

expr const &get_app_rev_args(expr const &e, buffer<expr> &args) {
    expr const *it = &e;
    while (is_app(*it)) {
        args.push_back(app_arg(*it));
        it = &(app_fn(*it));
    }
    return *it;
}

expr const &get_app_fn(expr const &e) {
    expr const *it = &e;
    while (is_app(*it)) {
        it = &(app_fn(*it));
    }
    return *it;
}

unsigned get_app_num_args(expr const &e) {
    expr const *it = &e;
    unsigned n = 0;
    while (is_app(*it)) {
        it = &(app_fn(*it));
        n++;
    }
    return n;
}

bool is_arrow(expr const &t) {
    if (!is_pi(t)) return false;
    if (has_loose_bvars(t)) {
        return !has_loose_bvar(binding_body(t), 0);
    } else {
        lean_assert(has_loose_bvars(binding_body(t)) ==
                    has_loose_bvar(binding_body(t), 0));
        return !has_loose_bvars(binding_body(t));
    }
}

bool is_default_var_name(name const &n) { return n == *g_default_name; }

// =======================================
// Update

expr update_mdata(expr const &e, expr const &t) {
    if (!is_eqp(mdata_expr(e), t))
        return mk_mdata(mdata_data(e), t);
    else
        return e;
}

expr update_proj(expr const &e, expr const &t) {
    if (!is_eqp(proj_expr(e), t))
        return mk_proj(proj_sname(e), proj_idx(e), t);
```

```
    else
        return e;
}

expr update_app(expr const &e, expr const &new_fn, expr const &new_arg) {
    if (!is_eqp(app_fn(e), new_fn) || !is_eqp(app_arg(e), new_arg))
        return mk_app(new_fn, new_arg);
    else
        return e;
}

expr update_binding(expr const &e, expr const &new_domain,
                    expr const &new_body) {
    if (!is_eqp(binding_domain(e), new_domain) ||
        !is_eqp(binding_body(e), new_body))
        return mk_binding(e.kind(), binding_name(e), new_domain, new_body,
                          binding_info(e));
    else
        return e;
}

expr update_binding(expr const &e, expr const &new_domain, expr const &new_body,
                    binder_info bi) {
    if (!is_eqp(binding_domain(e), new_domain) ||
        !is_eqp(binding_body(e), new_body) || bi != binding_info(e))
        return mk_binding(e.kind(), binding_name(e), new_domain, new_body, bi);
    else
        return e;
}

expr update_sort(expr const &e, level const &new_level) {
    if (!is_eqp(sort_level(e), new_level))
        return mk_sort(new_level);
    else
        return e;
}

expr update_const(expr const &e, levels const &new_levels) {
    if (!is_eqp(const_levels(e), new_levels))
        return mk_const(const_name(e), new_levels);
    else
        return e;
}

expr update_let(expr const &e, expr const &new_type, expr const &new_value,
                expr const &new_body) {
    if (!is_eqp(let_type(e), new_type) || !is_eqp(let_value(e), new_value) ||
        !is_eqp(let_body(e), new_body))
        return mk_let(let_name(e), new_type, new_value, new_body);
    else
        return e;
}

extern "C" object *lean_expr_update_mdata(obj_arg e, obj_arg new_expr) {
    if (mdata_expr(TO_REF(expr, e)).raw() != new_expr) {
        object *r = lean_expr_mk_mdata(mdata_data(TO_REF(expr, e)).to_obj_arg(),
                                       new_expr);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_expr);
        return e;
    }
}

extern "C" object *lean_expr_update_const(obj_arg e, obj_arg new_levels) {
    if (const_levels(TO_REF(expr, e)).raw() != new_levels) {
        object *r = lean_expr_mk_const(const_name(TO_REF(expr, e)).to_obj_arg(),
                                       new_levels);
        lean_dec_ref(e);
```

```
        return r;
    } else {
        lean_dec(new_levels);
        return e;
    }
}

extern "C" object *lean_expr_update_sort(obj_arg e, obj_arg new_level) {
    if (sort_level(TO_REF(expr, e)).raw() != new_level) {
        object *r = lean_expr_mk_sort(new_level);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec(new_level);
        return e;
    }
}

extern "C" object *lean_expr_update_proj(obj_arg e, obj_arg new_expr) {
    if (proj_expr(TO_REF(expr, e)).raw() != new_expr) {
        object *r =
            lean_expr_mk_proj(proj_sname(TO_REF(expr, e)).to_obj_arg(),
                              proj_idx(TO_REF(expr, e)).to_obj_arg(), new_expr);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_expr);
        return e;
    }
}

extern "C" object *lean_expr_update_app(obj_arg e, obj_arg new_fn,
                                        obj_arg new_arg) {
    if (app_fn(TO_REF(expr, e)).raw() != new_fn ||
        app_arg(TO_REF(expr, e)).raw() != new_arg) {
        object *r = lean_expr_mk_app(new_fn, new_arg);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_fn);
        lean_dec_ref(new_arg);
        return e;
    }
}

extern "C" object *lean_expr_update_forall(obj_arg e, uint8 new_binfo,
                                           obj_arg new_domain,
                                           obj_arg new_body) {
    if (binding_domain(TO_REF(expr, e)).raw() != new_domain ||
        binding_body(TO_REF(expr, e)).raw() != new_body ||
        binding_info(TO_REF(expr, e)) != static_cast<binder_info>(new_binfo)) {
        object *r =
            lean_expr_mk_forall(binding_name(TO_REF(expr, e)).to_obj_arg(),
                                new_domain, new_body, new_binfo);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_domain);
        lean_dec_ref(new_body);
        return e;
    }
}

extern "C" object *lean_expr_update_lambda(obj_arg e, uint8 new_binfo,
                                           obj_arg new_domain,
                                           obj_arg new_body) {
    if (binding_domain(TO_REF(expr, e)).raw() != new_domain ||
        binding_body(TO_REF(expr, e)).raw() != new_body ||
        binding_info(TO_REF(expr, e)) != static_cast<binder_info>(new_binfo)) {
        object *r =
```

```
            lean_expr_mk_lambda(binding_name(TO_REF(expr, e)).to_obj_arg(),
                                new_domain, new_body, new_binfo);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_domain);
        lean_dec_ref(new_body);
        return e;
    }
}

extern "C" object *lean_expr_update_let(obj_arg e, obj_arg new_type,
                                        obj_arg new_val, obj_arg new_body) {
    if (let_type(TO_REF(expr, e)).raw() != new_type ||
        let_value(TO_REF(expr, e)).raw() != new_val ||
        let_body(TO_REF(expr, e)).raw() != new_body) {
        object *r = lean_expr_mk_let(let_name(TO_REF(expr, e)).to_obj_arg(),
                                     new_type, new_val, new_body);
        lean_dec_ref(e);
        return r;
    } else {
        lean_dec_ref(new_type);
        lean_dec_ref(new_val);
        lean_dec_ref(new_body);
        return e;
    }
}

// =======================================
// Loose bound variable management

static bool has_loose_bvars_in_domain(expr const &b, unsigned vidx,
                                      bool strict) {
    if (is_pi(b)) {
        if (has_loose_bvar(binding_domain(b), vidx)) {
            if (is_explicit(binding_info(b))) {
                return true;
            } else if (has_loose_bvars_in_domain(binding_body(b), 0, strict)) {
                // "Transitivity": vidx occurs in current implicit argument, so
                // we search for current argument in the body.
                return true;
            }
        }
        // finally we search for vidx in the body
        return has_loose_bvars_in_domain(binding_body(b), vidx + 1, strict);
    } else if (!strict) {
        return has_loose_bvar(b, vidx);
    } else {
        return false;
    }
}

bool has_loose_bvar(expr const &e, unsigned i) {
    if (!has_loose_bvars(e)) return false;
    bool found = false;
    for_each(e, [&](expr const &e, unsigned offset) {
        if (found) return false;  // already found
        unsigned n_i = i + offset;
        if (n_i < i) return false;  // overflow, vidx can't be >= max unsigned
        if (n_i >= get_loose_bvar_range(e))
            return false;  // expression e does not contain bound variables with
                           // idx >= n_i
        if (is_var(e)) {
            nat const &vidx = bvar_idx(e);
            if (vidx == n_i) found = true;
        }
        return true;  // continue search
    });
    return found;
}
```

```
extern "C" uint8 lean_expr_has_loose_bvar(b_obj_arg e, b_obj_arg i) {
    if (!lean_is_scalar(i)) return false;
    return has_loose_bvar(TO_REF(expr, e), lean_unbox(i));
}

expr lower_loose_bvars(expr const &e, unsigned s, unsigned d) {
    if (d == 0 || s >= get_loose_bvar_range(e)) return e;
    lean_assert(s >= d);
    return replace(e, [=](expr const &e, unsigned offset) -> optional<expr> {
        unsigned s1 = s + offset;
        if (s1 < s)
            return some_expr(e);  // overflow, vidx can't be >= max unsigned
        if (s1 >= get_loose_bvar_range(e))
            return some_expr(e);  // expression e does not contain bound
                                  // variables with idx >= s1
        if (is_bvar(e) && bvar_idx(e) >= s1) {
            lean_assert(bvar_idx(e) >= offset + d);
            return some_expr(mk_bvar(bvar_idx(e) - nat(d)));
        } else {
            return none_expr();
        }
    });
}

expr lower_loose_bvars(expr const &e, unsigned d) {
    return lower_loose_bvars(e, d, d);
}

extern "C" object *lean_expr_lower_loose_bvars(b_obj_arg e, b_obj_arg s,
                                                b_obj_arg d) {
    if (!lean_is_scalar(s) || !lean_is_scalar(d) ||
        lean_unbox(s) < lean_unbox(d)) {
        lean_inc(e);
        return e;
    }
    return lower_loose_bvars(TO_REF(expr, e), lean_unbox(s), lean_unbox(d))
        .steal();
}

expr lift_loose_bvars(expr const &e, unsigned s, unsigned d) {
    if (d == 0 || s >= get_loose_bvar_range(e)) return e;
    return replace(e, [=](expr const &e, unsigned offset) -> optional<expr> {
        unsigned s1 = s + offset;
        if (s1 < s)
            return some_expr(e);  // overflow, vidx can't be >= max unsigned
        if (s1 >= get_loose_bvar_range(e))
            return some_expr(e);  // expression e does not contain bound
                                  // variables with idx >= s1
        if (is_var(e) && bvar_idx(e) >= s + offset) {
            return some_expr(mk_bvar(bvar_idx(e) + nat(d)));
        } else {
            return none_expr();
        }
    });
}

expr lift_loose_bvars(expr const &e, unsigned d) {
    return lift_loose_bvars(e, 0, d);
}

extern "C" object *lean_expr_lift_loose_bvars(b_obj_arg e, b_obj_arg s,
                                               b_obj_arg d) {
    if (!lean_is_scalar(s) || !lean_is_scalar(d)) {
        lean_inc(e);
        return e;
    }
    return lift_loose_bvars(TO_REF(expr, e), lean_unbox(s), lean_unbox(d))
        .steal();
}
```

```

// ========================================
// Implicit argument inference

expr infer_implicit(expr const &t, unsigned num_params, bool strict) {
    if (num_params == 0) {
        return t;
    } else if (is_pi(t)) {
        expr new_body = infer_implicit(binding_body(t), num_params - 1, strict);
        if (!is_explicit(binding_info(t))) {
            // argument is already marked as implicit
            return update_binding(t, binding_domain(t), new_body);
        } else if (has_loose_bvars_in_domain(new_body, 0, strict)) {
            return update_binding(t, binding_domain(t), new_body,
                                  mk_implicit_binder_info());
        } else {
            return update_binding(t, binding_domain(t), new_body);
        }
    } else {
        return t;
    }
}

expr infer_implicit(expr const &t, bool strict) {
    return infer_implicit(t, std::numeric_limits<unsigned>::max(), strict);
}

// ========================================
// Initialization & Finalization

void initialize_expr() {
    get_dummy();
    g_default_name = new name("a");
    mark_persistent(g_default_name->raw());
    g_Type0 = new expr(mk_sort(mk_level_one()));
    mark_persistent(g_Type0->raw());
    g_Prop = new expr(mk_sort(mk_level_zero()));
    mark_persistent(g_Prop->raw());
    /* TODO(Leo): add support for builtin constants in the kernel.
       Something similar to what we have in the library directory. */
}

void finalize_expr() {
    delete g_Prop;
    delete g_Type0;
    delete g_dummy;
    delete g_default_name;
}

// ========================================
// Legacy

optional<expr> has_expr_metavar_strict(expr const &e) {
    if (!has_expr_metavar(e)) return none_expr();
    optional<expr> r;
    for_each(e, [&](expr const &e, unsigned) {
        if (r || !has_expr_metavar(e)) return false;
        if (is_metavar_app(e)) {
            r = e;
            return false;
        }
        return true;
    });
    return r;
}
}  // namespace lean
// ::::::::::::::
// expr_eq_fn.cpp
// ::::::::::::::
/*
```

```cpp
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/interrupt.h>
#include <lean/thread.h>

#include <memory>
#include <vector>

#include "kernel/expr.h"
#include "kernel/expr_sets.h"

#ifndef LEAN_EQ_CACHE_CAPACITY
#define LEAN_EQ_CACHE_CAPACITY 1024 * 8
#endif

namespace lean {
struct eq_cache {
    struct entry {
        object *m_a;
        object *m_b;
        entry() : m_a(nullptr), m_b(nullptr) {}
    };
    unsigned m_capacity;
    std::vector<entry> m_cache;
    std::vector<unsigned> m_used;
    eq_cache()
        : m_capacity(LEAN_EQ_CACHE_CAPACITY), m_cache(LEAN_EQ_CACHE_CAPACITY) {}

    bool check(expr const &a, expr const &b) {
        if (!is_shared(a) || !is_shared(b)) return false;
        unsigned i = hash(hash(a), hash(b)) % m_capacity;
        if (m_cache[i].m_a == a.raw() && m_cache[i].m_b == b.raw()) {
            return true;
        } else {
            if (m_cache[i].m_a == nullptr) m_used.push_back(i);
            m_cache[i].m_a = a.raw();
            m_cache[i].m_b = b.raw();
            return false;
        }
    }

    void clear() {
        for (unsigned i : m_used) m_cache[i].m_a = nullptr;
        m_used.clear();
    }
};

/* CACHE_RESET: No */
MK_THREAD_LOCAL_GET_DEF(eq_cache, get_eq_cache);

/** \brief Functional object for comparing expressions.

    Remark if CompareBinderInfo is true, then functional object will also
   compare binder information attached to lambda and Pi expressions */
template <bool CompareBinderInfo>
class expr_eq_fn {
    eq_cache &m_cache;

    static void check_system() {
        ::lean::check_system("expression equality test");
    }

    bool apply(expr const &a, expr const &b) {
        if (is_eqp(a, b)) return true;
        if (hash(a) != hash(b)) return false;
        if (a.kind() != b.kind()) return false;
        if (is_bvar(a)) return bvar_idx(a) == bvar_idx(b);
```

```
        if (m_cache.check(a, b)) return true;
        switch (a.kind()) {
            case expr_kind::BVar:
                lean_unreachable();  // LCOV_EXCL_LINE
            case expr_kind::MData:
                return apply(mdata_expr(a), mdata_expr(b)) &&
                       mdata_data(a) == mdata_data(b);
            case expr_kind::Proj:
                return apply(proj_expr(a), proj_expr(b)) &&
                       proj_sname(a) == proj_sname(b) &&
                       proj_idx(a) == proj_idx(b);
            case expr_kind::Lit:
                return lit_value(a) == lit_value(b);
            case expr_kind::Const:
                return const_name(a) == const_name(b) &&
                       compare(const_levels(a), const_levels(b),
                               [](level const &l1, level const &l2) {
                                   return l1 == l2;
                               });
            case expr_kind::MVar:
                return mvar_name(a) == mvar_name(b);
            case expr_kind::FVar:
                return fvar_name(a) == fvar_name(b);
            case expr_kind::App:
                check_system();
                return apply(app_fn(a), app_fn(b)) &&
                       apply(app_arg(a), app_arg(b));
            case expr_kind::Lambda:
            case expr_kind::Pi:
                check_system();
                return apply(binding_domain(a), binding_domain(b)) &&
                       apply(binding_body(a), binding_body(b)) &&
                       (!CompareBinderInfo ||
                        binding_name(a) == binding_name(b)) &&
                       (!CompareBinderInfo ||
                        binding_info(a) == binding_info(b));
            case expr_kind::Let:
                check_system();
                return apply(let_type(a), let_type(b)) &&
                       apply(let_value(a), let_value(b)) &&
                       apply(let_body(a), let_body(b)) &&
                       (!CompareBinderInfo || let_name(a) == let_name(b));
            case expr_kind::Sort:
                return sort_level(a) == sort_level(b);
        }
        lean_unreachable();  // LCOV_EXCL_LINE
    }

   public:
    expr_eq_fn() : m_cache(get_eq_cache()) {}
    ~expr_eq_fn() { m_cache.clear(); }
    bool operator()(expr const &a, expr const &b) { return apply(a, b); }
};

bool is_equal(expr const &a, expr const &b) {
    return expr_eq_fn<false>()(a, b);
}
bool is_bi_equal(expr const &a, expr const &b) {
    return expr_eq_fn<true>()(a, b);
}

extern "C" uint8 lean_expr_eqv(b_obj_arg a, b_obj_arg b) {
    return expr_eq_fn<false>()(TO_REF(expr, a), TO_REF(expr, b));
}

extern "C" uint8 lean_expr_equal(b_obj_arg a, b_obj_arg b) {
    return expr_eq_fn<true>()(TO_REF(expr, a), TO_REF(expr, b));
}
}  // namespace lean
// ::::::::::::::
```

```cpp
// for_each_fn.cpp
// :::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/flet.h>
#include <lean/interrupt.h>
#include <lean/memory.h>

#include <utility>
#include <vector>

#include "kernel/cache_stack.h"
#include "kernel/for_each_fn.h"

#ifndef LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY
#define LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY 1024 * 8
#endif

namespace lean {
struct for_each_cache {
    struct entry {
        object const *m_cell;
        unsigned m_offset;
        entry() : m_cell(nullptr) {}
    };
    unsigned m_capacity;
    std::vector<entry> m_cache;
    std::vector<unsigned> m_used;
    for_each_cache(unsigned c) : m_capacity(c), m_cache(c) {}

    bool visited(expr const &e, unsigned offset) {
        unsigned i = hash(hash(e), offset) % m_capacity;
        if (m_cache[i].m_cell == e.raw() && m_cache[i].m_offset == offset) {
            return true;
        } else {
            if (m_cache[i].m_cell == nullptr) m_used.push_back(i);
            m_cache[i].m_cell = e.raw();
            m_cache[i].m_offset = offset;
            return false;
        }
    }

    void clear() {
        for (unsigned i : m_used) m_cache[i].m_cell = nullptr;
        m_used.clear();
    }
};

/* CACHE_RESET: NO */
MK_CACHE_STACK(for_each_cache, LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY)

class for_each_fn {
    for_each_cache_ref m_cache;
    std::function<bool(expr const &, unsigned)> m_f;  // NOLINT

    void apply(expr const &e, unsigned offset) {
        buffer<pair<expr const &, unsigned>> todo;
        todo.emplace_back(e, offset);
        while (true) {
        begin_loop:
            if (todo.empty()) break;
            check_interrupted();
            check_memory("expression traversal");
            auto p = todo.back();
            todo.pop_back();
            expr const &e = p.first;
```

```
            unsigned offset = p.second;

            switch (e.kind()) {
                case expr_kind::Const:
                case expr_kind::BVar:
                case expr_kind::Sort:
                    m_f(e, offset);
                    goto begin_loop;
                default:
                    break;
            }

            if (is_shared(e) && m_cache->visited(e, offset)) goto begin_loop;

            if (!m_f(e, offset)) goto begin_loop;

            switch (e.kind()) {
                case expr_kind::Const:
                case expr_kind::BVar:
                case expr_kind::Sort:
                case expr_kind::Lit:
                case expr_kind::MVar:
                case expr_kind::FVar:
                    goto begin_loop;
                case expr_kind::MData:
                    todo.emplace_back(mdata_expr(e), offset);
                    goto begin_loop;
                case expr_kind::Proj:
                    todo.emplace_back(proj_expr(e), offset);
                    goto begin_loop;
                case expr_kind::App:
                    todo.emplace_back(app_arg(e), offset);
                    todo.emplace_back(app_fn(e), offset);
                    goto begin_loop;
                case expr_kind::Lambda:
                case expr_kind::Pi:
                    todo.emplace_back(binding_body(e), offset + 1);
                    todo.emplace_back(binding_domain(e), offset);
                    goto begin_loop;
                case expr_kind::Let:
                    todo.emplace_back(let_body(e), offset + 1);
                    todo.emplace_back(let_value(e), offset);
                    todo.emplace_back(let_type(e), offset);
                    goto begin_loop;
            }
        }
    }

   public:
    for_each_fn(std::function<bool(expr const &, unsigned)> &&f)
        : m_f(f) {}  // NOLINT
    for_each_fn(std::function<bool(expr const &, unsigned)> const &f)
        : m_f(f) {}  // NOLINT
    void operator()(expr const &e) { apply(e, 0); }
};

void for_each(expr const &e,
              std::function<bool(expr const &, unsigned)> &&f) {  // NOLINT
    return for_each_fn(f)(e);
}
}  // namespace lean
// :::::::::::::
// inductive.cpp
// :::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
```

```cpp
#include <lean/sstream.h>
#include <lean/utf8.h>

#include "kernel/abstract.h"
#include "kernel/environment.h"
#include "kernel/find_fn.h"
#include "kernel/instantiate.h"
#include "kernel/kernel_exception.h"
#include "kernel/replace_fn.h"
#include "kernel/type_checker.h"
#include "util/name_generator.h"

namespace lean {
static name *g_ind_fresh = nullptr;

/**\ brief Return recursor name for the given inductive datatype name */
name mk_rec_name(name const &I) { return I + name("rec"); }

/** Return the names of all inductive datatypes */
static names get_all_inductive_names(buffer<inductive_type> const &ind_types) {
    buffer<name> all_names;
    for (inductive_type const &ind_type : ind_types) {
        all_names.push_back(ind_type.get_name());
    }
    return names(all_names);
}

/** Return the names of all inductive datatypes in the given inductive
 * declaration */
static names get_all_inductive_names(inductive_decl const &d) {
    buffer<inductive_type> ind_types;
    to_buffer(d.get_types(), ind_types);
    return get_all_inductive_names(ind_types);
}

/** \brief If \c d_name is the name of a non-empty inductive datatype, then
   return the name of the first constructor. Return none otherwise. */
static optional<name> get_first_cnstr(environment const &env,
                                      name const &d_name) {
    constant_info info = env.get(d_name);
    if (!info.is_inductive()) return optional<name>();
    names const &cnstrs = info.to_inductive_val().get_cnstrs();
    if (empty(cnstrs)) return optional<name>();
    return optional<name>(head(cnstrs));
}

optional<expr> mk_nullary_cnstr(environment const &env, expr const &type,
                                unsigned num_params) {
    buffer<expr> args;
    expr const &d = get_app_args(type, args);
    if (!is_constant(d)) return none_expr();
    name const &d_name = const_name(d);
    auto cnstr_name = get_first_cnstr(env, d_name);
    if (!cnstr_name) return none_expr();
    args.shrink(num_params);
    return some(mk_app(mk_constant(*cnstr_name, const_levels(d)), args));
}

optional<recursor_rule> get_rec_rule_for(recursor_val const &rec_val,
                                         expr const &major) {
    expr const &fn = get_app_fn(major);
    if (!is_constant(fn)) return optional<recursor_rule>();
    for (recursor_rule const &rule : rec_val.get_rules()) {
        if (rule.get_cnstr() == const_name(fn))
            return optional<recursor_rule>(rule);
    }
    return optional<recursor_rule>();
}

/* Auxiliary class for adding a mutual inductive datatype declaration. */
```

```
class add_inductive_fn {
    environment m_env;
    name_generator m_ngen;
    local_ctx m_lctx;
    names m_lparams;
    unsigned m_nparams;
    bool m_is_unsafe;
    buffer<inductive_type> m_ind_types;
    buffer<unsigned> m_nindices;
    level m_result_level;
    /* m_lparams ==> m_levels */
    levels m_levels;
    /* We track whether the resultant universe cannot be zero for any
       universe level instantiation */
    bool m_is_not_zero;
    /* A free variable for each parameter */
    buffer<expr> m_params;
    /* A constant for each inductive type */
    buffer<expr> m_ind_cnsts;

    level m_elim_level;
    bool m_K_target;

    bool m_is_nested;

    struct rec_info {
        expr m_C;              /* free variable for "main" motive */
        buffer<expr> m_minors; /* minor premises */
        buffer<expr> m_indices;
        expr m_major; /* major premise */
    };

    /* We have an entry for each inductive datatype being declared,
       and for nested inductive datatypes. */
    buffer<rec_info> m_rec_infos;

  public:
    add_inductive_fn(environment const &env, inductive_decl const &decl,
                     bool is_nested)
        : m_env(env),
          m_ngen(*g_ind_fresh),
          m_lparams(decl.get_lparams()),
          m_is_unsafe(decl.is_unsafe()),
          m_is_nested(is_nested) {
        if (!decl.get_nparams().is_small())
            throw kernel_exception(
                env,
                "invalid inductive datatype, number of parameters is too big");
        m_nparams = decl.get_nparams().get_small_value();
        to_buffer(decl.get_types(), m_ind_types);
    }

    type_checker tc() { return type_checker(m_env, m_lctx, !m_is_unsafe); }

    /** Return type of the parameter at position `i` */
    expr get_param_type(unsigned i) const {
        return m_lctx.get_local_decl(m_params[i]).get_type();
    }

    expr mk_local_decl(name const &n, expr const &t,
                       binder_info const &bi = binder_info()) {
        return m_lctx.mk_local_decl(m_ngen, n, t, bi);
    }

    expr mk_local_decl_for(expr const &t) {
        lean_assert(is_pi(t));
        return m_lctx.mk_local_decl(m_ngen, binding_name(t), binding_domain(t),
                                    binding_info(t));
    }

```

```
    expr whnf(expr const &t) { return tc().whnf(t); }

    expr infer_type(expr const &t) { return tc().infer(t); }

    bool is_def_eq(expr const &t1, expr const &t2) {
        return tc().is_def_eq(t1, t2);
    }

    expr mk_pi(buffer<expr> const &fvars, expr const &e) const {
        return m_lctx.mk_pi(fvars, e);
    }
    expr mk_pi(expr const &fvar, expr const &e) const {
        return m_lctx.mk_pi(1, &fvar, e);
    }
    expr mk_lambda(buffer<expr> const &fvars, expr const &e) const {
        return m_lctx.mk_lambda(fvars, e);
    }
    expr mk_lambda(expr const &fvar, expr const &e) const {
        return m_lctx.mk_lambda(1, &fvar, e);
    }

    /**
       \brief Check whether the type of each datatype is well typed, and do not
       contain free variables or meta variables, all inductive datatypes have
       the same parameters, the number of parameters match the argument
       m_nparams, and the result universes are equivalent.

       This method also initializes the fields:
       - m_levels
       - m_result_level
       - m_nindices
       - m_ind_cnsts
       - m_params

       \remark The local context m_lctx contains the free variables in m_params.
     */
    void check_inductive_types() {
        m_levels = lparams_to_levels(m_lparams);
        bool first = true;
        for (inductive_type const &ind_type : m_ind_types) {
            expr type = ind_type.get_type();
            m_env.check_name(ind_type.get_name());
            m_env.check_name(mk_rec_name(ind_type.get_name()));
            check_no_metavar_no_fvar(m_env, ind_type.get_name(), type);
            tc().check(type, m_lparams);
            m_nindices.push_back(0);
            unsigned i = 0;
            while (is_pi(type)) {
                if (i < m_nparams) {
                    if (first) {
                        expr param = mk_local_decl_for(type);
                        m_params.push_back(param);
                        type = instantiate(binding_body(type), param);
                    } else {
                        if (!is_def_eq(binding_domain(type), get_param_type(i)))
                            throw kernel_exception(
                                m_env,
                                "parameters of all inductive datatypes must "
                                "match");
                        type = instantiate(binding_body(type), m_params[i]);
                    }
                    i++;
                } else {
                    type = binding_body(type);
                    m_nindices.back()++;
                }
            }
            if (i != m_nparams)
                throw kernel_exception(m_env,
                                       "number of parameters mismatch in "
```

```
                                       "inductive datatype declaration");

            type = tc().ensure_sort(type);

            if (first) {
                m_result_level = sort_level(type);
                m_is_not_zero = is_not_zero(m_result_level);
            } else if (!is_equivalent(sort_level(type), m_result_level)) {
                throw kernel_exception(
                    m_env,
                    "mutually inductive types must live in the same universe");
            }

            m_ind_cnsts.push_back(mk_constant(ind_type.get_name(), m_levels));
            first = false;
        }

        lean_assert(length(m_levels) == length(m_lparams));
        lean_assert(m_nindices.size() == m_ind_types.size());
        lean_assert(m_ind_cnsts.size() == m_ind_types.size());
        lean_assert(m_params.size() == m_nparams);
    }

    /** \brief Return true if declaration is recursive */
    bool is_rec() {
        for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
            inductive_type const &ind_type = m_ind_types[idx];
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                expr t = constructor_type(cnstr);
                while (is_pi(t)) {
                    if (find(binding_domain(t), [&](expr const &e, unsigned) {
                            if (is_constant(e)) {
                                for (expr const &I : m_ind_cnsts)
                                    if (const_name(I) == const_name(e))
                                        return true;
                            }
                            return false;
                        })) {
                        return true;
                    }
                    t = binding_body(t);
                }
            }
        }
        return false;
    }

    /* Return true if the given declarataion is reflexive.

       Remark: We say an inductive type `T` is reflexive if it
       contains at least one constructor that takes as an argument a
       function returning `T'` where `T'` is another inductive datatype
       (possibly equal to `T`) in the same mutual declaration. */
    bool is_reflexive() {
        for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
            inductive_type const &ind_type = m_ind_types[idx];
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                expr t = constructor_type(cnstr);
                while (is_pi(t)) {
                    expr arg_type = binding_domain(t);
                    if (is_pi(arg_type) && has_ind_occ(arg_type)) return true;
                    expr local = mk_local_decl_for(t);
                    t = instantiate(binding_body(t), local);
                }
            }
        }
        return false;
    }

    /** Return list with the names of all inductive datatypes in the mutual
```

```
     * declaration. */
    names get_all_inductive_names() const {
        return ::lean::get_all_inductive_names(m_ind_types);
    }

    /** \brief Add all datatype declarations to environment. */
    void declare_inductive_types() {
        bool rec = is_rec();
        bool reflexive = is_reflexive();
        names all = get_all_inductive_names();
        for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
            inductive_type const &ind_type = m_ind_types[idx];
            name const &n = ind_type.get_name();
            buffer<name> cnstr_names;
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                cnstr_names.push_back(constructor_name(cnstr));
            }
            m_env.add_core(constant_info(
                inductive_val(n, m_lparams, ind_type.get_type(), m_nparams,
                              m_nindices[idx], all, names(cnstr_names), rec,
                              m_is_unsafe, reflexive, m_is_nested)));
        }
    }

    /** \brief Return true iff `t` is a term of the form `I As t`
        where `I` is the inductive datatype at position `i` being declared and
        `As` are the global parameters of this declaration. */
    bool is_valid_ind_app(expr const &t, unsigned i) {
        buffer<expr> args;
        expr I = get_app_args(t, args);
        if (I != m_ind_cnsts[i] || args.size() != m_nparams + m_nindices[i])
            return false;
        for (unsigned i = 0; i < m_nparams; i++) {
            if (m_params[i] != args[i]) return false;
        }
        return true;
    }

    /** \brief Return some(i) iff `t` is of the form `I As t` where `I` the
     * inductive `i`-th datatype being defined. */
    optional<unsigned> is_valid_ind_app(expr const &t) {
        for (unsigned i = 0; i < m_ind_types.size(); i++) {
            if (is_valid_ind_app(t, i)) return optional<unsigned>(i);
        }
        return optional<unsigned>();
    }

    /** \brief Return true iff `e` is one of the inductive datatype being
     * declared. */
    bool is_ind_occ(expr const &e) {
        return is_constant(e) &&
               std::any_of(m_ind_cnsts.begin(), m_ind_cnsts.end(),
                           [&](expr const &c) {
                               return const_name(e) == const_name(c);
                           });
    }

    /** \brief Return true iff `t` does not contain any occurrence of a datatype
     * being declared. */
    bool has_ind_occ(expr const &t) {
        return static_cast<bool>(
            find(t, [&](expr const &e, unsigned) { return is_ind_occ(e); }));
    }

    /** \brief Return `some(d_idx)` iff `t` is a recursive argument, `d_idx` is
       the index of the
        recursive inductive datatype. Otherwise, return `none`. */
    optional<unsigned> is_rec_argument(expr t) {
        t = whnf(t);
        while (is_pi(t)) {
```

```cpp
            expr local = mk_local_decl_for(t);
            t = whnf(instantiate(binding_body(t), local));
        }
        return is_valid_ind_app(t);
    }

    /** \brief Check if \c t contains only positive occurrences of the inductive
     * datatypes being declared. */
    void check_positivity(expr t, name const &cnstr_name, int arg_idx) {
        t = whnf(t);
        if (!has_ind_occ(t)) {
            // nonrecursive argument
        } else if (is_pi(t)) {
            if (has_ind_occ(binding_domain(t)))
                throw kernel_exception(
                    m_env, sstream() << "arg #" << (arg_idx + 1) << " of '"
                                     << cnstr_name
                                     << "' "
                                        "has a non positive occurrence of the "
                                        "datatypes being declared");
            expr local = mk_local_decl_for(t);
            check_positivity(instantiate(binding_body(t), local), cnstr_name,
                             arg_idx);
        } else if (is_valid_ind_app(t)) {
            // recursive argument
        } else {
            throw kernel_exception(
                m_env, sstream()
                           << "arg #" << (arg_idx + 1) << " of '" << cnstr_name
                           << "' "
                              "contains a non valid occurrence of the "
                              "datatypes being declared");
        }
    }

    /** \brief Check whether the constructor declarations are type correct,
       parameters are in the expected positions, constructor fields are in
       acceptable universe levels, positivity constraints, and returns the
       expected result. */
    void check_constructors() {
        for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
            inductive_type const &ind_type = m_ind_types[idx];
            name_set found_cnstrs;
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                name const &n = constructor_name(cnstr);
                if (found_cnstrs.contains(n)) {
                    throw kernel_exception(
                        m_env, sstream() << "duplicate constructor name '" << n
                                         << "'");
                }
                found_cnstrs.insert(n);
                expr t = constructor_type(cnstr);
                m_env.check_name(n);
                check_no_metavar_no_fvar(m_env, n, t);
                tc().check(t, m_lparams);
                unsigned i = 0;
                while (is_pi(t)) {
                    if (i < m_nparams) {
                        if (!is_def_eq(binding_domain(t), get_param_type(i)))
                            throw kernel_exception(
                                m_env, sstream() << "arg #" << (i + 1)
                                                 << " of '" << n << "' "
                                                 << "does not match inductive "
                                                    "datatypes parameters'");
                        t = instantiate(binding_body(t), m_params[i]);
                    } else {
                        expr s = tc().ensure_type(binding_domain(t));
                        // the sort is ok IF
                        //   1- its level is <= inductive datatype level, OR
                        //   2- is an inductive predicate
```

```
                        if (!(is_geq(m_result_level, sort_level(s)) ||
                              is_zero(m_result_level))) {
                            throw kernel_exception(
                                m_env,
                                sstream()
                                    << "universe level of type_of(arg #"
                                    << (i + 1) << ") "
                                    << "of '" << n
                                    << "' is too big for the corresponding "
                                       "inductive datatype");
                        }
                        if (!m_is_unsafe)
                            check_positivity(binding_domain(t), n, i);
                        expr local = mk_local_decl_for(t);
                        t = instantiate(binding_body(t), local);
                    }
                    i++;
                }
                if (!is_valid_ind_app(t, idx))
                    throw kernel_exception(
                        m_env, sstream()
                                   << "invalid return type for '" << n << "'");
            }
        }
    }

    void declare_constructors() {
        for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
            inductive_type const &ind_type = m_ind_types[idx];
            unsigned cidx = 0;
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                name const &n = constructor_name(cnstr);
                expr const &t = constructor_type(cnstr);
                unsigned arity = 0;
                expr it = t;
                while (is_pi(it)) {
                    it = binding_body(it);
                    arity++;
                }
                lean_assert(arity >= m_nparams);
                unsigned nfields = arity - m_nparams;
                m_env.add_core(constant_info(
                    constructor_val(n, m_lparams, t, ind_type.get_name(), cidx,
                                    m_nparams, nfields, m_is_unsafe)));
                cidx++;
            }
        }
    }

    /** \brief Return true if recursor can only map into Prop */
    bool elim_only_at_universe_zero() {
        if (m_is_not_zero) {
            /* For every universe parameter assignment, the resultant universe
               is not 0. So, it is not an inductive predicate */
            return false;
        }

        if (m_ind_types.size() > 1) {
            /* Mutually recursive inductive predicates only eliminate into Prop.
             */
            return true;
        }

        unsigned num_intros = length(m_ind_types[0].get_cnstrs());
        if (num_intros > 1) {
            /* We have more than one constructor, then recursor for inductive
               predicate can only eliminate intro Prop. */
            return true;
        }

```

```
        if (num_intros == 0) {
            /* empty inductive predicate (e.g., `false`) can eliminate into any
             * universe */
            return false;
        }

        /* We have only one constructor, the final check is, the type of each
           argument that is not a parameter: 1- It must live in Prop, *OR* 2- It
           must occur in the return type. (this is essentially what is called a
           non-uniform parameter in Coq). We can justify 2 by observing that
           this information is not a *secret* it is part of the type. By
           eliminating to a non-proposition, we would not be revealing anything
           that is not already known. */
        constructor const &cnstr = head(m_ind_types[0].get_cnstrs());
        expr type = constructor_type(cnstr);
        unsigned i = 0;
        buffer<expr> to_check; /* Arguments that we must check if occur in the
                                  result type */
        while (is_pi(type)) {
            expr fvar = mk_local_decl_for(type);
            if (i >= m_nparams) {
                expr s = tc().ensure_type(binding_domain(type));
                if (!is_zero(sort_level(s))) {
                    /* Current argument is not in Prop (i.e., condition 1
                       failed). We save it in to_check to be able to try
                       condition 2 above. */
                    to_check.push_back(fvar);
                }
            }
            type = instantiate(binding_body(type), fvar);
            i++;
        }
        buffer<expr> result_args;
        get_app_args(type, result_args);
        /* Check condition 2: every argument in to_check must occur in
         * result_args */
        for (expr const &arg : to_check) {
            if (std::find(result_args.begin(), result_args.end(), arg) ==
                result_args.end())
                return true; /* Condition 2 failed */
        }
        return false;
    }

    /** \brief Initialize m_elim_level. */
    void init_elim_level() {
        if (elim_only_at_universe_zero()) {
            m_elim_level = mk_level_zero();
        } else {
            name u("u");
            int i = 1;
            while (std::find(m_lparams.begin(), m_lparams.end(), u) !=
                   m_lparams.end()) {
                u = name("u").append_after(i);
                i++;
            }
            m_elim_level = mk_univ_param(u);
        }
    }

    void init_K_target() {
        /* A declaration is target for K-like reduction when
           it has one intro, the intro has 0 arguments, and it is an inductive
           predicate.
           In the following for-loop we check if the intro rule has 0 fields. */
        m_K_target =
            m_ind_types.size() ==
                1 && /* It is not a mutual declaration (for simplicity, we don't
                        gain anything by supporting K in mutual declarations. */
            is_zero(m_result_level) && /* It is an inductive predicate. */
```

```
            length(m_ind_types[0].get_cnstrs()) ==
                1; /* Inductive datatype has only one constructor. */
        if (!m_K_target) return;
        expr it = constructor_type(head(m_ind_types[0].get_cnstrs()));
        unsigned i = 0;
        while (is_pi(it)) {
            if (i < m_nparams) {
                it = binding_body(it);
            } else {
                /* See comment above */
                m_K_target = false;
                break;
            }
            i++;
        }
    }

    /** \brief Given `t` of the form `I As is` where `I` is one of the inductive
       datatypes being defined, As are the global parameters, and is the actual
       indices provided to it. Return the index of `I`, and store is in the
       argument `indices`. */
    unsigned get_I_indices(expr const &t, buffer<expr> &indices) {
        optional<unsigned> r = is_valid_ind_app(t);
        lean_assert(r);
        buffer<expr> all_args;
        get_app_args(t, all_args);
        for (unsigned i = m_nparams; i < all_args.size(); i++)
            indices.push_back(all_args[i]);
        return *r;
    }

    /** \brief Populate m_rec_infos. */
    void mk_rec_infos() {
        unsigned d_idx = 0;
        /* First, populate the fields, m_C, m_indices, m_major */
        for (inductive_type const &ind_type : m_ind_types) {
            rec_info info;
            expr t = ind_type.get_type();
            unsigned i = 0;
            while (is_pi(t)) {
                if (i < m_nparams) {
                    t = instantiate(binding_body(t), m_params[i]);
                } else {
                    expr idx = mk_local_decl_for(t);
                    info.m_indices.push_back(idx);
                    t = instantiate(binding_body(t), idx);
                }
                i++;
            }
            info.m_major = mk_local_decl(
                "t",
                mk_app(mk_app(m_ind_cnsts[d_idx], m_params), info.m_indices));
            expr C_ty = mk_sort(m_elim_level);
            C_ty = mk_pi(info.m_major, C_ty);
            C_ty = mk_pi(info.m_indices, C_ty);
            name C_name("motive");
            if (m_ind_types.size() > 1)
                C_name = name(C_name).append_after(d_idx + 1);
            info.m_C = mk_local_decl(C_name, C_ty);
            m_rec_infos.push_back(info);
            d_idx++;
        }
        /* First, populate the field m_minors */
        unsigned minor_idx = 1;
        d_idx = 0;
        for (inductive_type const &ind_type : m_ind_types) {
            name ind_type_name = ind_type.get_name();
            for (constructor const &cnstr : ind_type.get_cnstrs()) {
                buffer<expr> b_u;  // nonrec and rec args;
                buffer<expr> u;    // rec args
```

```
                buffer<expr> v;    // inductive args
                name cnstr_name = constructor_name(cnstr);
                expr t = constructor_type(cnstr);
                unsigned i = 0;
                while (is_pi(t)) {
                    if (i < m_nparams) {
                        t = instantiate(binding_body(t), m_params[i]);
                    } else {
                        expr l = mk_local_decl_for(t);
                        b_u.push_back(l);
                        if (is_rec_argument(binding_domain(t))) u.push_back(l);
                        t = instantiate(binding_body(t), l);
                    }
                    i++;
                }
                buffer<expr> it_indices;
                unsigned it_idx = get_I_indices(t, it_indices);
                expr C_app = mk_app(m_rec_infos[it_idx].m_C, it_indices);
                expr intro_app = mk_app(
                    mk_app(mk_constant(cnstr_name, m_levels), m_params), b_u);
                C_app = mk_app(C_app, intro_app);
                /* populate v using u */
                for (unsigned i = 0; i < u.size(); i++) {
                    expr u_i = u[i];
                    expr u_i_ty = whnf(infer_type(u_i));
                    buffer<expr> xs;
                    while (is_pi(u_i_ty)) {
                        expr x = mk_local_decl_for(u_i_ty);
                        xs.push_back(x);
                        u_i_ty = whnf(instantiate(binding_body(u_i_ty), x));
                    }
                    buffer<expr> it_indices;
                    unsigned it_idx = get_I_indices(u_i_ty, it_indices);
                    expr C_app = mk_app(m_rec_infos[it_idx].m_C, it_indices);
                    expr u_app = mk_app(u_i, xs);
                    C_app = mk_app(C_app, u_app);
                    expr v_i_ty = mk_pi(xs, C_app);
                    expr v_i = mk_local_decl(name("v").append_after(i), v_i_ty,
                                             binder_info());
                    v.push_back(v_i);
                }
                expr minor_ty = mk_pi(b_u, mk_pi(v, C_app));
                name minor_name =
                    cnstr_name.replace_prefix(ind_type_name, name());
                expr minor = mk_local_decl(minor_name, minor_ty);
                m_rec_infos[d_idx].m_minors.push_back(minor);
                minor_idx++;
            }
            d_idx++;
        }
    }

    /** \brief Return the levels for the recursor. */
    levels get_rec_levels() {
        if (is_param(m_elim_level))
            return levels(m_elim_level, m_levels);
        else
            return m_levels;
    }

    /** \brief Return the level parameter names for the recursor. */
    names get_rec_lparams() {
        if (is_param(m_elim_level))
            return names(param_id(m_elim_level), m_lparams);
        else
            return m_lparams;
    }

    /** \brief Store all type formers in `Cs` */
    void collect_Cs(buffer<expr> &Cs) {
```

```
        for (unsigned i = 0; i < m_ind_types.size(); i++)
            Cs.push_back(m_rec_infos[i].m_C);
    }

    /** \brief Store all minor premises in `ms`. */
    void collect_minor_premises(buffer<expr> &ms) {
        for (unsigned i = 0; i < m_ind_types.size(); i++)
            ms.append(m_rec_infos[i].m_minors);
    }

    recursor_rules mk_rec_rules(unsigned d_idx, buffer<expr> const &Cs,
                                buffer<expr> const &minors,
                                unsigned &minor_idx) {
        inductive_type const &d = m_ind_types[d_idx];
        levels lvls = get_rec_levels();
        buffer<recursor_rule> rules;
        for (constructor const &cnstr : d.get_cnstrs()) {
            buffer<expr> b_u;
            buffer<expr> u;
            expr t = constructor_type(cnstr);
            unsigned i = 0;
            while (is_pi(t)) {
                if (i < m_nparams) {
                    t = instantiate(binding_body(t), m_params[i]);
                } else {
                    expr l = mk_local_decl_for(t);
                    b_u.push_back(l);
                    if (is_rec_argument(binding_domain(t))) u.push_back(l);
                    t = instantiate(binding_body(t), l);
                }
                i++;
            }
            buffer<expr> v;
            for (unsigned i = 0; i < u.size(); i++) {
                expr u_i = u[i];
                expr u_i_ty = whnf(infer_type(u_i));
                buffer<expr> xs;
                while (is_pi(u_i_ty)) {
                    expr x = mk_local_decl_for(u_i_ty);
                    xs.push_back(x);
                    u_i_ty = whnf(instantiate(binding_body(u_i_ty), x));
                }
                buffer<expr> it_indices;
                unsigned it_idx = get_I_indices(u_i_ty, it_indices);
                name rec_name = mk_rec_name(m_ind_types[it_idx].get_name());
                expr rec_app = mk_constant(rec_name, lvls);
                rec_app = mk_app(
                    mk_app(
                        mk_app(mk_app(mk_app(rec_app, m_params), Cs), minors),
                        it_indices),
                    mk_app(u_i, xs));
                v.push_back(mk_lambda(xs, rec_app));
            }
            expr e_app = mk_app(mk_app(minors[minor_idx], b_u), v);
            expr comp_rhs = mk_lambda(
                m_params,
                mk_lambda(Cs, mk_lambda(minors, mk_lambda(b_u, e_app))));
            rules.push_back(
                recursor_rule(constructor_name(cnstr), b_u.size(), comp_rhs));
            minor_idx++;
        }
        return recursor_rules(rules);
    }

    /** \brief Declare recursors. */
    void declare_recursors() {
        buffer<expr> Cs;
        collect_Cs(Cs);
        buffer<expr> minors;
        collect_minor_premises(minors);
```

```
        unsigned nminors = minors.size();
        unsigned nmotives = Cs.size();
        names all = get_all_inductive_names();
        unsigned minor_idx = 0;
        for (unsigned d_idx = 0; d_idx < m_ind_types.size(); d_idx++) {
            rec_info const &info = m_rec_infos[d_idx];
            expr C_app = mk_app(mk_app(info.m_C, info.m_indices), info.m_major);
            expr rec_ty = mk_pi(info.m_major, C_app);
            rec_ty = mk_pi(info.m_indices, rec_ty);
            rec_ty = mk_pi(minors, rec_ty);
            rec_ty = mk_pi(Cs, rec_ty);
            rec_ty = mk_pi(m_params, rec_ty);
            rec_ty = infer_implicit(rec_ty, true /* strict */);
            recursor_rules rules = mk_rec_rules(d_idx, Cs, minors, minor_idx);
            name rec_name = mk_rec_name(m_ind_types[d_idx].get_name());
            names rec_lparams = get_rec_lparams();
            m_env.add_core(constant_info(
                recursor_val(rec_name, rec_lparams, rec_ty, all, m_nparams,
                             m_nindices[d_idx], nmotives, nminors, rules,
                             m_K_target, m_is_unsafe)));
        }
    }

    environment operator()() {
        m_env.check_duplicated_univ_params(m_lparams);
        check_inductive_types();
        declare_inductive_types();
        check_constructors();
        declare_constructors();
        init_elim_level();
        init_K_target();
        mk_rec_infos();
        declare_recursors();
        return m_env;
    }
};

static name *g_nested = nullptr;
static name *g_nested_fresh = nullptr;

/* Result produced by elim_nested_inductive_fn */
struct elim_nested_inductive_result {
    name_generator m_ngen;
    buffer<expr> m_params;
    name_map<expr> m_aux2nested; /* mapping from auxiliary type to nested
                                    inductive type. */
    declaration m_aux_decl;

    elim_nested_inductive_result(name_generator const &ngen,
                                 buffer<expr> const &params,
                                 buffer<pair<expr, name>> const &nested_aux,
                                 declaration const &d)
        : m_ngen(ngen), m_params(params), m_aux_decl(d) {
        for (pair<expr, name> const &p : nested_aux) {
            m_aux2nested.insert(p.second, p.first);
        }
    }

    /* If `c` is an constructor name associated with an auxiliary inductive
       type, then return the
       nested inductive associated with it and the name of its inductive type.
       Return none. */
    optional<pair<expr, name>> get_nested_if_aux_constructor(
        environment const &aux_env, name const &c) const {
        optional<constant_info> info = aux_env.find(c);
        if (!info || !info->is_constructor())
            return optional<pair<expr, name>>();
        name auxI_name = info->to_constructor_val().get_induct();
        expr const *nested = m_aux2nested.find(auxI_name);
        if (!nested) return optional<pair<expr, name>>();
```

```
        return optional<pair<expr, name>>(*nested, auxI_name);
    }

    name restore_constructor_name(environment const &aux_env,
                                  name const &cnstr_name) const {
        optional<pair<expr, name>> p =
            get_nested_if_aux_constructor(aux_env, cnstr_name);
        lean_assert(p);
        expr const &I = get_app_fn(p->first);
        lean_assert(is_constant(I));
        return cnstr_name.replace_prefix(p->second, const_name(I));
    }

    expr restore_nested(
        expr e, environment const &aux_env,
        name_map<name> const &aux_rec_name_map = name_map<name>()) {
        local_ctx lctx;
        buffer<expr> As;
        bool pi = is_pi(e);
        for (unsigned i = 0; i < m_params.size(); i++) {
            lean_assert(is_pi(e) || is_lambda(e));
            As.push_back(lctx.mk_local_decl(
                m_ngen, binding_name(e), binding_domain(e), binding_info(e)));
            e = instantiate(binding_body(e), As.back());
        }
        e = replace(e, [&](expr const &t, unsigned) {
            if (is_constant(t)) {
                if (name const *rec_name =
                        aux_rec_name_map.find(const_name(t))) {
                    return some_expr(mk_constant(*rec_name, const_levels(t)));
                }
            }
            expr const &fn = get_app_fn(t);
            if (is_constant(fn)) {
                if (expr const *nested = m_aux2nested.find(const_name(fn))) {
                    buffer<expr> args;
                    get_app_args(t, args);
                    lean_assert(args.size() >= m_params.size());
                    expr new_t = instantiate_rev(
                        abstract(*nested, m_params.size(), m_params.data()),
                        As.size(), As.data());
                    return some_expr(mk_app(new_t,
                                            args.size() - m_params.size(),
                                            args.data() + m_params.size()));
                }
                if (optional<pair<expr, name>> r =
                        get_nested_if_aux_constructor(aux_env,
                                                      const_name(fn))) {
                    expr nested = r->first;
                    name auxI_name = r->second;
                    /* `t` is a constructor-application of an auxiliary
                     * inductive type */
                    buffer<expr> args;
                    get_app_args(t, args);
                    lean_assert(args.size() >= m_params.size());
                    expr new_nested = instantiate_rev(
                        abstract(nested, m_params.size(), m_params.data()),
                        As.size(), As.data());
                    buffer<expr> I_args;
                    expr I = get_app_args(new_nested, I_args);
                    lean_assert(is_constant(I));
                    name new_fn_name =
                        const_name(fn).replace_prefix(auxI_name, const_name(I));
                    expr new_fn = mk_constant(new_fn_name, const_levels(I));
                    expr new_t = mk_app(mk_app(new_fn, I_args),
                                        args.size() - m_params.size(),
                                        args.data() + m_params.size());
                    return some_expr(new_t);
                }
            }
```

```
            return none_expr();
        });
        return pi ? lctx.mk_pi(As, e) : lctx.mk_lambda(As, e);
    }
};

/* Eliminate nested inductive datatypes by creating a new (auxiliary)
   declaration which contains and inductive types in `d` and copies of the
   nested inductive datatypes used in `d`. For each nested occurrence `I Ds is`
   where `I` is a nested inductive datatype and `Ds` are the parametric
   arguments and `is` the indices, we create an auxiliary type `Iaux` in the
   (mutual) inductive declaration `d`, and replace `I Ds is` with `Iaux As is`
   where `As` are `d`'s parameters. Moreover, we add the pair `(I Ds, Iaux)` to
   `nested_aux`.

   Note that, `As` and `Ds` may have a different sizes. */
struct elim_nested_inductive_fn {
    environment const &m_env;
    declaration const &m_d;
    name_generator m_ngen;
    local_ctx m_params_lctx;
    buffer<expr> m_params;
    buffer<pair<expr, name>>
        m_nested_aux; /* The expressions stored here contains free vars in
                         `m_params` */
    levels m_lvls;
    buffer<inductive_type> m_new_types;
    unsigned m_next_idx{1};

    elim_nested_inductive_fn(environment const &env, declaration const &d)
        : m_env(env), m_d(d), m_ngen(*g_nested_fresh) {
        m_lvls = lparams_to_levels(inductive_decl(m_d).get_lparams());
    }

    name mk_unique_name(name const &n) {
        while (true) {
            name r = n.append_after(m_next_idx);
            m_next_idx++;
            if (!m_env.find(r)) return r;
        }
    }

    void throw_ill_formed() {
        throw kernel_exception(
            m_env, "invalid nested inductive datatype, ill-formed declaration");
    }

    expr replace_params(expr const &e, buffer<expr> const &As) {
        lean_assert(m_params.size() == As.size());
        return instantiate_rev(abstract(e, As.size(), As.data()),
                               m_params.size(), m_params.data());
    }

    /* IF `e` is of the form `I Ds is` where
          1) `I` is a nested inductive datatype (i.e., a previously declared
       inductive datatype), 2) the parametric arguments `Ds` do not contain
       loose bound variables, and do contain inductive datatypes in
       `m_new_types` THEN return the `inductive_val` in the `constant_info`
       associated with `I`. Otherwise, return none. */
    optional<inductive_val> is_nested_inductive_app(expr const &e) {
        if (!is_app(e)) return optional<inductive_val>();
        expr const &fn = get_app_fn(e);
        if (!is_constant(fn)) return optional<inductive_val>();
        optional<constant_info> info = m_env.find(const_name(fn));
        if (!info || !info->is_inductive()) return optional<inductive_val>();
        buffer<expr> args;
        get_app_args(e, args);
        unsigned nparams = info->to_inductive_val().get_nparams();
        if (nparams > args.size()) return optional<inductive_val>();
        bool is_nested = false;
```

```
        bool loose_bvars = false;
        for (unsigned i = 0; i < nparams; i++) {
            if (has_loose_bvars(args[i])) {
                loose_bvars = true;
            }
            if (find(args[i], [&](expr const &t, unsigned) {
                    if (is_constant(t)) {
                        for (inductive_type const &ind_type : m_new_types) {
                            if (const_name(t) == ind_type.get_name())
                                return true;
                        }
                    }
                    return false;
                })) {
                is_nested = true;
            }
        }
        if (!is_nested) return optional<inductive_val>();
        if (loose_bvars)
            throw kernel_exception(
                m_env, sstream() << "invalid nested inductive datatype '"
                                 << const_name(fn)
                                 << "', nested inductive datatypes parameters "
                                    "cannot contain local variables.");
        return optional<inductive_val>(info->to_inductive_val());
    }

    expr instantiate_pi_params(expr e, unsigned nparams, expr const *params) {
        for (unsigned i = 0; i < nparams; i++) {
            if (!is_pi(e)) throw_ill_formed();
            e = binding_body(e);
        }
        return instantiate_rev(e, nparams, params);
    }

    /* If `e` is a nested occurrence `I Ds is`, return `Iaux As is` */
    optional<expr> replace_if_nested(local_ctx const &lctx,
                                     buffer<expr> const &As, expr const &e) {
        optional<inductive_val> I_val = is_nested_inductive_app(e);
        if (!I_val) return none_expr();
        /* `e` is of the form `I As is` where `As` are the parameters and `is`
         * the indices */
        buffer<expr> args;
        expr const &fn = get_app_args(e, args);
        name const &I_name = const_name(fn);
        levels const &I_lvls = const_levels(fn);
        lean_assert(I_val->get_nparams() <= args.size());
        unsigned I_nparams = I_val->get_nparams();
        expr IAs = mk_app(fn, I_nparams, args.data()); /* `I As` */
        /* Check whether we have already created an auxiliary inductive_type for
         * `I As` */
        optional<name> auxI_name;
        /* Replace `As` with `m_params` before searching at `m_nested_aux`.
           We need this step because we re-create parameters for each
           constructor with the correct binding info */
        expr Iparams = replace_params(IAs, As);
        for (pair<expr, name> const &p : m_nested_aux) {
            /* Remark: we could have used `is_def_eq` here instead of structural
               equality. It is probably not needed, but if one day we decide to
               do it, we have to populate an auxiliary environment with the
               inductive datatypes we are defining since `p.first` and `Iparams`
               contain references to them. */
            if (p.first == Iparams) {
                auxI_name = p.second;
                break;
            }
        }
        if (auxI_name) {
            expr auxI = mk_constant(*auxI_name, m_lvls);
            auxI = mk_app(auxI, As);
```

```
            return some_expr(
                mk_app(auxI, args.size() - I_nparams, args.data() + I_nparams));
        } else {
            optional<expr> result;
            /* We should copy all inductive datatypes `J` in the mutual
               declaration containing `I` to the `m_new_types` mutual
               declaration as new auxiliary types. */
            for (name const &J_name : I_val->get_all()) {
                constant_info J_info = m_env.get(J_name);
                lean_assert(J_info.is_inductive());
                expr J = mk_constant(J_name, I_lvls);
                expr JAs = mk_app(J, I_nparams, args.data());
                name auxJ_name = mk_unique_name(*g_nested + J_name);
                expr auxJ_type = instantiate_lparams(
                    J_info.get_type(), J_info.get_lparams(), I_lvls);
                auxJ_type =
                    instantiate_pi_params(auxJ_type, I_nparams, args.data());
                auxJ_type = lctx.mk_pi(As, auxJ_type);
                m_nested_aux.push_back(
                    mk_pair(replace_params(JAs, As), auxJ_name));
                if (J_name == I_name) {
                    /* Create result */
                    expr auxI = mk_constant(auxJ_name, m_lvls);
                    auxI = mk_app(auxI, As);
                    result = mk_app(auxI, args.size() - I_nparams,
                                    args.data() + I_nparams);
                }
                buffer<constructor> auxJ_constructors;
                for (name const &J_cnstr_name :
                     J_info.to_inductive_val().get_cnstrs()) {
                    constant_info J_cnstr_info = m_env.get(J_cnstr_name);
                    name auxJ_cnstr_name =
                        J_cnstr_name.replace_prefix(J_name, auxJ_name);
                    /* auxJ_cnstr_type still has references to `J`, this will be
                     * fixed later when we process it. */
                    expr auxJ_cnstr_type =
                        instantiate_lparams(J_cnstr_info.get_type(),
                                            J_cnstr_info.get_lparams(), I_lvls);
                    auxJ_cnstr_type = instantiate_pi_params(
                        auxJ_cnstr_type, I_nparams, args.data());
                    auxJ_cnstr_type = lctx.mk_pi(As, auxJ_cnstr_type);
                    auxJ_constructors.push_back(
                        constructor(auxJ_cnstr_name, auxJ_cnstr_type));
                }
                m_new_types.push_back(inductive_type(
                    auxJ_name, auxJ_type, constructors(auxJ_constructors)));
            }
            lean_assert(result);
            return result;
        }
    }

    /* Replace all nested inductive datatype occurrences in `e`. */
    expr replace_all_nested(local_ctx const &lctx, buffer<expr> const &As,
                            expr const &e) {
        return replace(e, [&](expr const &e, unsigned) {
            return replace_if_nested(lctx, As, e);
        });
    }

    expr get_params(expr type, unsigned nparams, local_ctx &lctx,
                    buffer<expr> &params) {
        lean_assert(params.empty());
        for (unsigned i = 0; i < nparams; i++) {
            if (!is_pi(type))
                throw kernel_exception(
                    m_env,
                    "invalid inductive datatype declaration, incorrect number "
                    "of parameters");
            params.push_back(lctx.mk_local_decl(m_ngen, binding_name(type),
```

```
                                                binding_domain(type),
                                                binding_info(type)));
            type = instantiate(binding_body(type), params.back());
        }
        return type;
    }

    elim_nested_inductive_result operator()() {
        inductive_decl ind_d(m_d);
        if (!ind_d.get_nparams().is_small()) throw_ill_formed();
        unsigned d_nparams = ind_d.get_nparams().get_small_value();
        to_buffer(ind_d.get_types(), m_new_types);
        if (m_new_types.size() == 0)
            throw kernel_exception(
                m_env,
                "invalid empty (mutual) inductive datatype declaration, it "
                "must contain at least one inductive type.");
        /* initialize m_params and m_params_lctx */
        get_params(m_new_types[0].get_type(), d_nparams, m_params_lctx,
                   m_params);
        unsigned qhead = 0;
        /* Main elimination loop. */
        while (qhead < m_new_types.size()) {
            inductive_type ind_type = m_new_types[qhead];
            buffer<constructor> new_cnstrs;
            for (constructor cnstr : ind_type.get_cnstrs()) {
                expr cnstr_type = constructor_type(cnstr);
                local_ctx lctx;
                buffer<expr> As;
                /* Consume parameters.

                   We (re-)create the parameters for each constructor because we
                   want to preserve the binding_info. */
                cnstr_type = get_params(cnstr_type, d_nparams, lctx, As);
                lean_assert(As.size() == d_nparams);
                expr new_cnstr_type = replace_all_nested(lctx, As, cnstr_type);
                new_cnstr_type = lctx.mk_pi(As, new_cnstr_type);
                new_cnstrs.push_back(
                    constructor(constructor_name(cnstr), new_cnstr_type));
            }
            m_new_types[qhead] =
                inductive_type(ind_type.get_name(), ind_type.get_type(),
                               constructors(new_cnstrs));
            qhead++;
        }
        declaration aux_decl =
            mk_inductive_decl(ind_d.get_lparams(), ind_d.get_nparams(),
                              inductive_types(m_new_types), ind_d.is_unsafe());
        return elim_nested_inductive_result(m_ngen, m_params, m_nested_aux,
                                            aux_decl);
    }
};

/* Given the auxiliary environment `aux_env` generated by processing the
   auxiliary mutual declaration, and the original declaration `d`. This function
   return a pair `(aux_rec_names, aux_rec_name_map)` where `aux_rec_names`
   contains the recursor names associated to auxiliary inductive types used to
   eliminated nested inductive occurrences.
   The mapping `aux_rec_name_map` contains an entry `(aux_rec_name -> rec_name)`
   for each element in `aux_rec_names`. It provides the new names for these
   recursors.

   We compute the new recursor names using the first inductive datatype in the
   original declaration `d`, and the suffice `.rec_<idx>`. */
static pair<names, name_map<name>> mk_aux_rec_name_map(
    environment const &aux_env, inductive_decl const &d) {
    unsigned ntypes = length(d.get_types());
    lean_assert(ntypes > 0);
    inductive_type const &main_type = head(d.get_types());
    name const &main_name = main_type.get_name();
```

```
    constant_info main_info = aux_env.get(main_name);
    names const &all_names = main_info.to_inductive_val().get_all();
    /* This function is only called if we have created auxiliary inductive types
       when eliminating the nested inductives. */
    lean_assert(length(all_names) > ntypes);
    /* Remark: we use the `main_name` to declarate the auxiliary recursors as:
       <main_name>.rec_1, <main_name>.rec_2, ... This is a little bit
       asymmetrical if `d` is a mutual declaration, but it makes sure we have
       simple names. */
    buffer<name> old_rec_names;
    name_map<name> rec_map;
    unsigned i = 0;
    unsigned next_idx = 1;
    for (name const &ind_name : all_names) {
        if (i >= ntypes) {
            old_rec_names.push_back(mk_rec_name(ind_name));
            name new_rec_name = mk_rec_name(main_name).append_after(next_idx);
            next_idx++;
            rec_map.insert(old_rec_names.back(), new_rec_name);
        }
        i++;
    }
    return mk_pair(names(old_rec_names), rec_map);
}

environment environment::add_inductive(declaration const &d) const {
    elim_nested_inductive_result res = elim_nested_inductive_fn(*this, d)();
    bool is_nested = !res.m_aux2nested.empty();
    environment aux_env =
        add_inductive_fn(*this, inductive_decl(res.m_aux_decl), is_nested)();
    if (!is_nested) {
        /* `d` did not contain nested inductive types. */
        return aux_env;
    } else {
        /* Restore nested inductives. */
        inductive_decl ind_d(d);
        names all_ind_names = get_all_inductive_names(ind_d);
        names aux_rec_names;
        name_map<name> aux_rec_name_map;
        std::tie(aux_rec_names, aux_rec_name_map) =
            mk_aux_rec_name_map(aux_env, d);
        environment new_env = *this;
        auto process_rec = [&](name const &rec_name) {
            name new_rec_name = rec_name;
            if (name const *new_name = aux_rec_name_map.find(rec_name))
                new_rec_name = *new_name;
            constant_info rec_info = aux_env.get(rec_name);
            expr new_rec_type = res.restore_nested(rec_info.get_type(), aux_env,
                                                   aux_rec_name_map);
            recursor_val rec_val = rec_info.to_recursor_val();
            buffer<recursor_rule> new_rules;
            for (recursor_rule const &rule : rec_val.get_rules()) {
                expr new_rhs = res.restore_nested(rule.get_rhs(), aux_env,
                                                  aux_rec_name_map);
                name cnstr_name = rule.get_cnstr();
                name new_cnstr_name = cnstr_name;
                if (new_rec_name != rec_name) {
                    /* We need to fix the constructor name */
                    new_cnstr_name =
                        res.restore_constructor_name(aux_env, cnstr_name);
                }
                new_rules.push_back(
                    recursor_rule(new_cnstr_name, rule.get_nfields(), new_rhs));
            }
            new_env.check_name(new_rec_name);
            new_env.add_core(constant_info(
                recursor_val(new_rec_name, rec_info.get_lparams(), new_rec_type,
                             all_ind_names, rec_val.get_nparams(),
                             rec_val.get_nindices(), rec_val.get_nmotives(),
                             rec_val.get_nminors(), recursor_rules(new_rules),
```

```
                                rec_val.is_k(), rec_val.is_unsafe())));
        };
        for (inductive_type const &ind_type : ind_d.get_types()) {
            constant_info ind_info = aux_env.get(ind_type.get_name());
            inductive_val ind_val = ind_info.to_inductive_val();
            /* We just need to "fix" the `all` fields for ind_info.

               Remark: if we decide to store the recursor names, we will also
               need to fix it. */
            new_env.add_core(constant_info(inductive_val(
                ind_info.get_name(), ind_info.get_lparams(),
                ind_info.get_type(), ind_val.get_nparams(),
                ind_val.get_nindices(), all_ind_names, ind_val.get_cnstrs(),
                ind_val.is_rec(), ind_val.is_unsafe(), ind_val.is_reflexive(),
                ind_val.is_nested())));
            for (name const &cnstr_name : ind_val.get_cnstrs()) {
                constant_info cnstr_info = aux_env.get(cnstr_name);
                constructor_val cnstr_val = cnstr_info.to_constructor_val();
                expr new_type =
                    res.restore_nested(cnstr_info.get_type(), aux_env);
                new_env.add_core(constant_info(constructor_val(
                    cnstr_info.get_name(), cnstr_info.get_lparams(), new_type,
                    cnstr_val.get_induct(), cnstr_val.get_cidx(),
                    cnstr_val.get_nparams(), cnstr_val.get_nfields(),
                    cnstr_val.is_unsafe())));
            }
            process_rec(mk_rec_name(ind_type.get_name()));
        }
        for (name const &aux_rec : aux_rec_names) {
            process_rec(aux_rec);
        }
        return new_env;
    }
}

static expr *g_nat_zero = nullptr;
static expr *g_nat_succ = nullptr;
static expr *g_string_mk = nullptr;
static expr *g_list_cons_char = nullptr;
static expr *g_list_nil_char = nullptr;
static expr *g_char_of_nat = nullptr;

expr nat_lit_to_constructor(expr const &e) {
    lean_assert(is_nat_lit(e));
    nat const &v = lit_value(e).get_nat();
    if (v == 0u)
        return *g_nat_zero;
    else
        return mk_app(*g_nat_succ, mk_lit(literal(v - nat(1))));
}

expr string_lit_to_constructor(expr const &e) {
    lean_assert(is_string_lit(e));
    string_ref const &s = lit_value(e).get_string();
    std::vector<unsigned> cs;
    utf8_decode(s.to_std_string(), cs);
    expr r = *g_list_nil_char;
    unsigned i = cs.size();
    while (i > 0) {
        i--;
        r = mk_app(*g_list_cons_char,
                   mk_app(*g_char_of_nat, mk_lit(literal(cs[i]))), r);
    }
    return mk_app(*g_string_mk, r);
}

void initialize_inductive() {
    g_nested = new name("_nested");
    mark_persistent(g_nested->raw());
    g_ind_fresh = new name("_ind_fresh");
```

```
    mark_persistent(g_ind_fresh->raw());
    g_nested_fresh = new name("_nested_fresh");
    mark_persistent(g_nested_fresh->raw());
    g_nat_zero = new expr(mk_constant(name{"Nat", "zero"}));
    mark_persistent(g_nat_zero->raw());
    g_nat_succ = new expr(mk_constant(name{"Nat", "succ"}));
    mark_persistent(g_nat_succ->raw());
    g_string_mk = new expr(mk_constant(name{"String", "mk"}));
    mark_persistent(g_string_mk->raw());
    expr char_type = mk_constant(name{"Char"});
    g_list_cons_char = new expr(
        mk_app(mk_constant(name{"List", "cons"}, {level()}), char_type));
    mark_persistent(g_list_cons_char->raw());
    g_list_nil_char = new expr(
        mk_app(mk_constant(name{"List", "nil"}, {level()}), char_type));
    mark_persistent(g_list_nil_char->raw());
    g_char_of_nat = new expr(mk_constant(name{"Char", "ofNat"}));
    mark_persistent(g_char_of_nat->raw());
    register_name_generator_prefix(*g_ind_fresh);
    register_name_generator_prefix(*g_nested_fresh);
}

void finalize_inductive() {
    delete g_nested;
    delete g_ind_fresh;
    delete g_nested_fresh;
    delete g_nat_succ;
    delete g_nat_zero;
    delete g_string_mk;
    delete g_list_cons_char;
    delete g_list_nil_char;
}
}  // namespace lean
// ::::::::::::::
// init_module.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include "kernel/declaration.h"
#include "kernel/environment.h"
#include "kernel/expr.h"
#include "kernel/inductive.h"
#include "kernel/level.h"
#include "kernel/local_ctx.h"
#include "kernel/quot.h"
#include "kernel/type_checker.h"

namespace lean {
void initialize_kernel_module() {
    initialize_level();
    initialize_expr();
    initialize_declaration();
    initialize_type_checker();
    initialize_environment();
    initialize_local_ctx();
    initialize_inductive();
    initialize_quot();
}

void finalize_kernel_module() {
    finalize_quot();
    finalize_inductive();
    finalize_local_ctx();
    finalize_environment();
    finalize_type_checker();
    finalize_declaration();
```

```cpp
    finalize_expr();
    finalize_level();
}
}  // namespace lean
// ::::::::::::::
// instantiate.cpp
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <algorithm>
#include <limits>

#include "kernel/declaration.h"
#include "kernel/instantiate.h"
#include "kernel/replace_fn.h"

namespace lean {
expr instantiate(expr const &a, unsigned s, unsigned n, expr const *subst) {
    if (s >= get_loose_bvar_range(a) || n == 0) return a;
    return replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
        unsigned s1 = s + offset;
        if (s1 < s)
            return some_expr(m);  // overflow, vidx can't be >= max unsigned
        if (s1 >= get_loose_bvar_range(m))
            return some_expr(m);  // expression m does not contain loose bound
                                  // variables with idx >= s1
        if (is_bvar(m)) {
            nat const &vidx = bvar_idx(m);
            if (vidx >= s1) {
                unsigned h = s1 + n;
                if (h < s1 /* overflow, h is bigger than any vidx */ ||
                    vidx < h) {
                    return some_expr(lift_loose_bvars(
                        subst[vidx.get_small_value() - s1], offset));
                } else {
                    return some_expr(mk_bvar(vidx - nat(n)));
                }
            }
        }
        return none_expr();
    });
}

expr instantiate(expr const &e, unsigned n, expr const *s) {
    return instantiate(e, 0, n, s);
}
expr instantiate(expr const &e, std::initializer_list<expr> const &l) {
    return instantiate(e, l.size(), l.begin());
}
expr instantiate(expr const &e, unsigned i, expr const &s) {
    return instantiate(e, i, 1, &s);
}
expr instantiate(expr const &e, expr const &s) { return instantiate(e, 0, s); }

extern "C" object *lean_expr_instantiate1(object *a0, object *e0) {
    expr const &a = reinterpret_cast<expr const &>(a0);
    if (!has_loose_bvars(a)) {
        lean_inc(a0);
        return a0;
    }
    expr const &e = reinterpret_cast<expr const &>(e0);
    expr r = instantiate(a, 1, &e);
    return r.steal();
}

static object *lean_expr_instantiate_core(b_obj_arg a0, size_t n,
```

```
                                          object **subst) {
    expr const &a = reinterpret_cast<expr const &>(a0);
    if (!has_loose_bvars(a) || n == 0) {
        lean_inc(a0);
        return a0;
    }
    expr r = replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
        if (offset >= get_loose_bvar_range(m))
            return some_expr(m);  // expression m does not contain loose bound
                                  // variables with idx >= offset
        if (is_bvar(m)) {
            nat const &vidx = bvar_idx(m);
            if (vidx >= offset) {
                size_t h = offset + n;
                if (h < offset /* overflow, h is bigger than any vidx */ ||
                    (vidx.is_small() && vidx.get_small_value() < h)) {
                    object *v = subst[vidx.get_small_value() - offset];
                    return some_expr(lift_loose_bvars(TO_REF(expr, v), offset));
                } else {
                    return some_expr(mk_bvar(vidx - nat::of_size_t(n)));
                }
            }
        }
        return none_expr();
    });
    return r.steal();
}

extern "C" object *lean_expr_instantiate(b_obj_arg a, b_obj_arg subst) {
    return lean_expr_instantiate_core(a, lean_array_size(subst),
                                      lean_array_cptr(subst));
}

extern "C" object *lean_expr_instantiate_range(b_obj_arg a, b_obj_arg begin,
                                               b_obj_arg end, b_obj_arg subst) {
    if (!lean_is_scalar(begin) || !lean_is_scalar(end)) {
        lean_internal_panic("invalid range for Expr.instantiateRange");
    } else {
        usize sz = lean_array_size(subst);
        usize b = lean_unbox(begin);
        usize e = lean_unbox(end);
        if (b > e || e > sz) {
            lean_internal_panic("invalid range for Expr.instantiateRange");
        }
        return lean_expr_instantiate_core(a, e - b, lean_array_cptr(subst) + b);
    }
}

expr instantiate_rev(expr const &a, unsigned n, expr const *subst) {
    if (!has_loose_bvars(a)) return a;
    return replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
        if (offset >= get_loose_bvar_range(m))
            return some_expr(m);  // expression m does not contain loose bound
                                  // variables with idx >= offset
        if (is_bvar(m)) {
            nat const &vidx = bvar_idx(m);
            if (vidx >= offset) {
                size_t h = offset + n;
                if (h < offset /* overflow, h is bigger than any vidx */ ||
                    (vidx.is_small() && vidx.get_small_value() < h)) {
                    return some_expr(lift_loose_bvars(
                        subst[n - (vidx.get_small_value() - offset) - 1],
                        offset));
                } else {
                    return some_expr(mk_bvar(vidx - nat(n)));
                }
            }
        }
        return none_expr();
    });
```

```
}

static object *lean_expr_instantiate_rev_core(object *a0, size_t n,
                                              object **subst) {
    expr const &a = reinterpret_cast<expr const &>(a0);
    if (!has_loose_bvars(a)) {
        lean_inc(a0);
        return a0;
    }
    expr r = replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
        if (offset >= get_loose_bvar_range(m))
            return some_expr(m);  // expression m does not contain loose bound
                                  // variables with idx >= offset
        if (is_bvar(m)) {
            nat const &vidx = bvar_idx(m);
            if (vidx >= offset) {
                size_t h = offset + n;
                if (h < offset /* overflow, h is bigger than any vidx */ ||
                    (vidx.is_small() && vidx.get_small_value() < h)) {
                    object *v =
                        subst[n - (vidx.get_small_value() - offset) - 1];
                    return some_expr(lift_loose_bvars(TO_REF(expr, v), offset));
                } else {
                    return some_expr(mk_bvar(vidx - nat::of_size_t(n)));
                }
            }
        }
        return none_expr();
    });
    return r.steal();
}

extern "C" object *lean_expr_instantiate_rev(b_obj_arg a, b_obj_arg subst) {
    return lean_expr_instantiate_rev_core(a, lean_array_size(subst),
                                          lean_array_cptr(subst));
}

extern "C" object *lean_expr_instantiate_rev_range(b_obj_arg a, b_obj_arg begin,
                                                   b_obj_arg end,
                                                   b_obj_arg subst) {
    if (!lean_is_scalar(begin) || !lean_is_scalar(end)) {
        lean_internal_panic("invalid range for Expr.instantiateRevRange");
    } else {
        usize sz = lean_array_size(subst);
        usize b = lean_unbox(begin);
        usize e = lean_unbox(end);
        if (b > e || e > sz) {
            lean_internal_panic("invalid range for Expr.instantiateRevRange");
        }
        return lean_expr_instantiate_rev_core(a, e - b,
                                              lean_array_cptr(subst) + b);
    }
}

bool is_head_beta(expr const &t) {
    return is_app(t) && is_lambda(get_app_fn(t));
}

expr apply_beta(expr f, unsigned num_args, expr const *args) {
    if (num_args == 0) {
        return f;
    } else if (!is_lambda(f)) {
        return mk_rev_app(f, num_args, args);
    } else {
        unsigned m = 1;
        while (is_lambda(binding_body(f)) && m < num_args) {
            f = binding_body(f);
            m++;
        }
        lean_assert(m <= num_args);
```

```
        return mk_rev_app(
            instantiate(binding_body(f), m, args + (num_args - m)),
            num_args - m, args);
    }
}

expr head_beta_reduce(expr const &t) {
    if (!is_head_beta(t)) {
        return t;
    } else {
        buffer<expr> args;
        expr const &f = get_app_rev_args(t, args);
        lean_assert(is_lambda(f));
        return head_beta_reduce(apply_beta(f, args.size(), args.data()));
    }
}

expr cheap_beta_reduce(expr const &e) {
    if (!is_app(e)) return e;
    expr fn = get_app_fn(e);
    if (!is_lambda(fn)) return e;
    buffer<expr> args;
    get_app_args(e, args);
    unsigned i = 0;
    while (is_lambda(fn) && i < args.size()) {
        i++;
        fn = binding_body(fn);
    }
    if (!has_loose_bvars(fn)) {
        return mk_app(fn, args.size() - i, args.data() + i);
    } else if (is_bvar(fn)) {
        lean_assert(bvar_idx(fn) < i);
        return mk_app(args[i - bvar_idx(fn).get_small_value() - 1],
                      args.size() - i, args.data() + i);
    } else {
        return e;
    }
}

expr instantiate_lparams(expr const &e, names const &lps, levels const &ls) {
    if (!has_param_univ(e)) return e;
    return replace(e, [&](expr const &e) -> optional<expr> {
        if (!has_param_univ(e)) return some_expr(e);
        if (is_constant(e)) {
            return some_expr(update_constant(
                e, map_reuse(const_levels(e), [&](level const &l) {
                    return instantiate(l, lps, ls);
                })));
        } else if (is_sort(e)) {
            return some_expr(
                update_sort(e, instantiate(sort_level(e), lps, ls)));
        } else {
            return none_expr();
        }
    });
}

expr instantiate_type_lparams(constant_info const &info, levels const &ls) {
    if (info.get_num_lparams() != length(ls))
        lean_internal_panic(
            "#universes mismatch at instantiateTypeLevelParams");
    if (is_nil(ls) || !has_param_univ(info.get_type())) return info.get_type();
    return instantiate_lparams(info.get_type(), info.get_lparams(), ls);
}

expr instantiate_value_lparams(constant_info const &info, levels const &ls) {
    if (info.get_num_lparams() != length(ls))
        lean_internal_panic(
            "#universes mismatch at instantiateValueLevelParams");
    if (!info.has_value())
```

```cpp
        lean_internal_panic(
            "definition/theorem expected at instantiateValueLevelParams");
    if (is_nil(ls) || !has_param_univ(info.get_value()))
        return info.get_value();
    return instantiate_lparams(info.get_value(), info.get_lparams(), ls);
}

extern "C" object *lean_instantiate_type_lparams(b_obj_arg info, b_obj_arg ls) {
    return instantiate_type_lparams(TO_REF(constant_info, info),
                                    TO_REF(levels, ls))
        .steal();
}

extern "C" object *lean_instantiate_value_lparams(b_obj_arg info,
                                                  b_obj_arg ls) {
    return instantiate_value_lparams(TO_REF(constant_info, info),
                                     TO_REF(levels, ls))
        .steal();
}
}  // namespace lean
// ::::::::::::::
// level.cpp
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/debug.h>
#include <lean/hash.h>
#include <lean/interrupt.h>

#include <algorithm>
#include <unordered_set>
#include <utility>
#include <vector>

#include "kernel/environment.h"
#include "kernel/level.h"
#include "util/buffer.h"
#include "util/list.h"

namespace lean {

extern "C" usize lean_level_hash(obj_arg l);
extern "C" unsigned lean_level_depth(obj_arg l);
extern "C" uint8 lean_level_has_mvar(obj_arg l);
extern "C" uint8 lean_level_has_param(obj_arg l);

extern "C" object *lean_level_mk_zero(object *);
extern "C" object *lean_level_mk_succ(obj_arg);
extern "C" object *lean_level_mk_mvar(obj_arg);
extern "C" object *lean_level_mk_param(obj_arg);
extern "C" object *lean_level_mk_max(obj_arg, obj_arg);
extern "C" object *lean_level_mk_imax(obj_arg, obj_arg);
extern "C" object *lean_level_mk_max_simp(obj_arg, obj_arg);
extern "C" object *lean_level_mk_imax_simp(obj_arg, obj_arg);

level mk_succ(level const &l) {
    return level(lean_level_mk_succ(l.to_obj_arg()));
}
level mk_max_core(level const &l1, level const &l2) {
    return level(lean_level_mk_max(l1.to_obj_arg(), l2.to_obj_arg()));
}
level mk_imax_core(level const &l1, level const &l2) {
    return level(lean_level_mk_imax(l1.to_obj_arg(), l2.to_obj_arg()));
}
level mk_univ_param(name const &n) {
    return level(lean_level_mk_param(n.to_obj_arg()));
```

```cpp
}
level mk_univ_mvar(name const &n) {
    return level(lean_level_mk_mvar(n.to_obj_arg()));
}

unsigned level::hash() const { return lean_level_hash(to_obj_arg()); }
unsigned get_depth(level const &l) { return lean_level_depth(l.to_obj_arg()); }
bool has_param(level const &l) { return lean_level_has_param(l.to_obj_arg()); }
bool has_mvar(level const &l) { return lean_level_has_mvar(l.to_obj_arg()); }

bool is_explicit(level const &l) {
    switch (kind(l)) {
        case level_kind::Zero:
            return true;
        case level_kind::Param:
        case level_kind::MVar:
        case level_kind::Max:
        case level_kind::IMax:
            return false;
        case level_kind::Succ:
            return is_explicit(succ_of(l));
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

/** \brief Convert (succ^k l) into (l, k). If l is not a succ, then return (l,
 * 0) */
pair<level, unsigned> to_offset(level l) {
    unsigned k = 0;
    while (is_succ(l)) {
        l = succ_of(l);
        k++;
    }
    return mk_pair(l, k);
}

unsigned to_explicit(level const &l) {
    lean_assert(is_explicit(l));
    return to_offset(l).second;
}

level mk_max(level const &l1, level const &l2) {
    if (is_explicit(l1) && is_explicit(l2)) {
        return get_depth(l1) >= get_depth(l2) ? l1 : l2;
    } else if (l1 == l2) {
        return l1;
    } else if (is_zero(l1)) {
        return l2;
    } else if (is_zero(l2)) {
        return l1;
    } else if (is_max(l2) && (max_lhs(l2) == l1 || max_rhs(l2) == l1)) {
        return l2;  // if l2 == (max l1 l'), then max l1 l2 == l2
    } else if (is_max(l1) && (max_lhs(l1) == l2 || max_rhs(l1) == l2)) {
        return l1;  // if l1 == (max l2 l'), then max l1 l2 == l1
    } else {
        auto p1 = to_offset(l1);
        auto p2 = to_offset(l2);
        if (p1.first == p2.first) {
            lean_assert(p1.second != p2.second);
            return p1.second > p2.second ? l1 : l2;
        } else {
            return mk_max_core(l1, l2);
        }
    }
}

level mk_imax(level const &l1, level const &l2) {
    if (is_not_zero(l2))
        return mk_max(l1, l2);
    else if (is_zero(l2))
```

```cpp
        return l2;  // imax u 0 = 0  for any u
    else if (is_zero(l1))
        return l2;  // imax 0 u = u  for any u
    else if (l1 == l2)
        return l1;  // imax u u = u
    else
        return mk_imax_core(l1, l2);
}

static level *g_level_zero = nullptr;
static level *g_level_one = nullptr;
level const &mk_level_zero() { return *g_level_zero; }
level const &mk_level_one() { return *g_level_one; }
bool is_one(level const &l) { return l == mk_level_one(); }

bool operator==(level const &l1, level const &l2) {
    if (kind(l1) != kind(l2)) return false;
    if (hash(l1) != hash(l2)) return false;
    if (is_eqp(l1, l2)) return true;
    switch (kind(l1)) {
        case level_kind::Zero:
            return true;
        case level_kind::Param:
        case level_kind::MVar:
            return level_id(l1) == level_id(l2);
        case level_kind::Max:
        case level_kind::IMax:
        case level_kind::Succ:
            if (get_depth(l1) != get_depth(l2)) return false;
            break;
    }
    switch (kind(l1)) {
        case level_kind::Zero:
        case level_kind::Param:
        case level_kind::MVar:
            lean_unreachable();  // LCOV_EXCL_LINE
        case level_kind::Max:
        case level_kind::IMax:
            return level_lhs(l1) == level_lhs(l2) &&
                   level_rhs(l1) == level_rhs(l2);
        case level_kind::Succ:
            return succ_of(l1) == succ_of(l2);
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

extern "C" uint8 lean_level_eqv(object *l1, object *l2) {
    return is_equivalent(TO_REF(level, l1), TO_REF(level, l2));
}

extern "C" uint8 lean_level_eq(object *l1, object *l2) {
    return TO_REF(level, l1) == TO_REF(level, l2);
}

bool is_not_zero(level const &l) {
    switch (kind(l)) {
        case level_kind::Zero:
        case level_kind::Param:
        case level_kind::MVar:
            return false;
        case level_kind::Succ:
            return true;
        case level_kind::Max:
            return is_not_zero(max_lhs(l)) || is_not_zero(max_rhs(l));
        case level_kind::IMax:
            return is_not_zero(imax_rhs(l));
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}
```

```
bool is_lt(level const &a, level const &b, bool use_hash) {
    if (is_eqp(a, b)) return false;
    unsigned da = get_depth(a);
    unsigned db = get_depth(b);
    if (da < db) return true;
    if (da > db) return false;
    if (kind(a) != kind(b)) return kind(a) < kind(b);
    if (use_hash) {
        if (hash(a) < hash(b)) return true;
        if (hash(a) > hash(b)) return false;
    }
    if (a == b) return false;
    switch (kind(a)) {
        case level_kind::Zero:
            lean_unreachable();  // LCOV_EXCL_LINE
        case level_kind::Param:
        case level_kind::MVar:
            return level_id(a) < level_id(b);
        case level_kind::Max:
        case level_kind::IMax:
            if (level_lhs(a) != level_lhs(b))
                return is_lt(level_lhs(a), level_lhs(b), use_hash);
            else
                return is_lt(level_rhs(a), level_rhs(b), use_hash);
        case level_kind::Succ:
            return is_lt(succ_of(a), succ_of(b), use_hash);
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

bool is_lt(levels const &as, levels const &bs, bool use_hash) {
    if (is_nil(as)) return !is_nil(bs);
    if (is_nil(bs)) return false;
    if (car(as) == car(bs))
        return is_lt(cdr(as), cdr(bs), use_hash);
    else
        return is_lt(car(as), car(bs), use_hash);
}

bool levels_has_param(b_obj_arg ls) {
    while (!is_scalar(ls)) {
        if (lean_level_has_param(cnstr_get(ls, 0))) return true;
        ls = cnstr_get(ls, 1);
    }
    return false;
}

bool levels_has_mvar(b_obj_arg ls) {
    while (!is_scalar(ls)) {
        if (lean_level_has_mvar(cnstr_get(ls, 0))) return true;
        ls = cnstr_get(ls, 1);
    }
    return false;
}

bool has_param(levels const &ls) { return levels_has_param(ls.raw()); }
bool has_mvar(levels const &ls) { return levels_has_mvar(ls.raw()); }

void for_each_level_fn::apply(level const &l) {
    if (!m_f(l)) return;
    switch (l.kind()) {
        case level_kind::Succ:
            apply(succ_of(l));
            break;
        case level_kind::Max:
        case level_kind::IMax:
            apply(level_lhs(l));
            apply(level_rhs(l));
            break;
        case level_kind::Zero:
```

```
        case level_kind::Param:
        case level_kind::MVar:
            break;
    }
}

level replace_level_fn::apply(level const &l) {
    optional<level> r = m_f(l);
    if (r) return *r;
    switch (l.kind()) {
        case level_kind::Succ:
            return update_succ(l, apply(succ_of(l)));
        case level_kind::Max:
        case level_kind::IMax: {
            level l1 = apply(level_lhs(l));
            level l2 = apply(level_rhs(l));
            return update_max(l, l1, l2);
        }
        case level_kind::Zero:
        case level_kind::Param:
        case level_kind::MVar:
            return l;
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

bool occurs(level const &u, level const &l) {
    bool found = false;
    for_each(l, [&](level const &l) {
        if (found) return false;
        if (l == u) {
            found = true;
            return false;
        }
        return true;
    });
    return found;
}

optional<name> get_undef_param(level const &l, names const &ps) {
    optional<name> r;
    for_each(l, [&](level const &l) {
        if (!has_param(l) || r) return false;
        if (is_param(l) &&
            std::find(ps.begin(), ps.end(), param_id(l)) == ps.end())
            r = param_id(l);
        return true;
    });
    return r;
}

level update_succ(level const &l, level const &new_arg) {
    if (is_eqp(succ_of(l), new_arg))
        return l;
    else
        return mk_succ(new_arg);
}

level update_max(level const &l, level const &new_lhs, level const &new_rhs) {
    if (is_eqp(level_lhs(l), new_lhs) && is_eqp(level_rhs(l), new_rhs))
        return l;
    else if (is_max(l))
        return mk_max(new_lhs, new_rhs);
    else
        return mk_imax(new_lhs, new_rhs);
}

extern "C" object *lean_level_update_succ(obj_arg l, obj_arg new_arg) {
    if (succ_of(TO_REF(level, l)).raw() == new_arg) {
        lean_dec(new_arg);
```

```
        return l;
    } else {
        lean_dec_ref(l);
        return lean_level_mk_succ(new_arg);
    }
}

extern "C" object *lean_level_update_max(obj_arg l, obj_arg new_lhs,
                                         obj_arg new_rhs) {
    if (max_lhs(TO_REF(level, l)).raw() == new_lhs &&
        max_rhs(TO_REF(level, l)).raw() == new_rhs) {
        lean_dec(new_lhs);
        lean_dec(new_rhs);
        return l;
    } else {
        lean_dec_ref(l);
        return lean_level_mk_max_simp(new_lhs, new_rhs);
    }
}

extern "C" object *lean_level_update_imax(obj_arg l, obj_arg new_lhs,
                                          obj_arg new_rhs) {
    if (imax_lhs(TO_REF(level, l)).raw() == new_lhs &&
        imax_rhs(TO_REF(level, l)).raw() == new_rhs) {
        lean_dec(new_lhs);
        lean_dec(new_rhs);
        return l;
    } else {
        lean_dec_ref(l);
        return lean_level_mk_imax_simp(new_lhs, new_rhs);
    }
}

level instantiate(level const &l, names const &ps, levels const &ls) {
    lean_assert(length(ps) == length(ls));
    return replace(l, [=](level const &l) {
        if (!has_param(l)) {
            return some_level(l);
        } else if (is_param(l)) {
            name const &id = param_id(l);
            names const *it1 = &ps;
            levels const *it2 = &ls;
            /* The assertion above ensures that !is_nil(*it2) is unnecessay, but
               we we keep it here to ensure the lean_instantiate_lparams does
               not crash at runtime when misused. */
            while (!is_nil(*it1) && !is_nil(*it2)) {
                 if (head(*it1) == id) return some_level(head(*it2));
                 it1 = &tail(*it1);
                 it2 = &tail(*it2);
            }
            return some_level(l);
        } else {
            return none_level();
        }
    });
}

static void print(std::ostream &out, level l);

static void print_child(std::ostream &out, level const &l) {
    if (is_explicit(l) || is_param(l) || is_mvar(l)) {
        print(out, l);
    } else {
        out << "(";
        print(out, l);
        out << ")";
    }
}

static void print(std::ostream &out, level l) {
```

```
    if (is_explicit(l)) {
        out << get_depth(l);
    } else {
        switch (kind(l)) {
            case level_kind::Zero:
                lean_unreachable();  // LCOV_EXCL_LINE
            case level_kind::Param:
                out << param_id(l);
                break;
            case level_kind::MVar:
                out << "?" << mvar_id(l);
                break;
            case level_kind::Succ:
                out << "succ ";
                print_child(out, succ_of(l));
                break;
            case level_kind::Max:
            case level_kind::IMax:
                if (is_max(l))
                    out << "max ";
                else
                    out << "imax ";
                print_child(out, level_lhs(l));
                // max and imax are right associative
                while (kind(level_rhs(l)) == kind(l)) {
                    l = level_rhs(l);
                    out << " ";
                    print_child(out, level_lhs(l));
                }
                out << " ";
                print_child(out, level_rhs(l));
                break;
        }
    }
}

std::ostream &operator<<(std::ostream &out, level const &l) {
    print(out, l);
    return out;
}

format pp(level l, bool unicode, unsigned indent);

static format pp_child(level const &l, bool unicode, unsigned indent) {
    if (is_explicit(l) || is_param(l) || is_mvar(l)) {
        return pp(l, unicode, indent);
    } else {
        return paren(pp(l, unicode, indent));
    }
}

format pp(level l, bool unicode, unsigned indent) {
    if (is_explicit(l)) {
        return format(get_depth(l));
    } else {
        switch (kind(l)) {
            case level_kind::Zero:
                lean_unreachable();  // LCOV_EXCL_LINE
            case level_kind::Param:
                return format(param_id(l));
            case level_kind::MVar:
                return format("?") + format(mvar_id(l));
            case level_kind::Succ: {
                auto p = to_offset(l);
                auto fmt1 = pp_child(p.first, unicode, indent);
                return fmt1 + format("+") + format(p.second);
            }
            case level_kind::Max:
            case level_kind::IMax: {
                format r = format(is_max(l) ? "max" : "imax");
```

```
                r += nest(indent, compose(line(), pp_child(level_lhs(l),
                                                          unicode, indent)));
                // max and imax are right associative
                while (kind(level_rhs(l)) == kind(l)) {
                    l = level_rhs(l);
                    r += nest(indent,
                              compose(line(),
                                      pp_child(level_lhs(l), unicode, indent)));
                }
                r += nest(indent, compose(line(), pp_child(level_rhs(l),
                                                          unicode, indent)));
                return group(r);
            }
        }
        lean_unreachable();  // LCOV_EXCL_LINE
    }
}

format pp(level const &l, options const &opts) {
    return pp(l, get_pp_unicode(opts), get_pp_indent(opts));
}

format pp(level const &lhs, level const &rhs, bool unicode, unsigned indent) {
    format leq = unicode ? format("≤") : format("<=");
    return group(pp(lhs, unicode, indent) + space() + leq + line() +
                 pp(rhs, unicode, indent));
}

format pp(level const &lhs, level const &rhs, options const &opts) {
    return pp(lhs, rhs, get_pp_unicode(opts), get_pp_indent(opts));
}

// A total order on level expressions that has the following properties
//  - succ(l) is an immediate successor of l.
//  - zero is the minimal element.
// This total order is used in the normalization procedure.
static bool is_norm_lt(level const &a, level const &b) {
    if (is_eqp(a, b)) return false;
    auto p1 = to_offset(a);
    auto p2 = to_offset(b);
    level const &l1 = p1.first;
    level const &l2 = p2.first;
    if (l1 != l2) {
        if (kind(l1) != kind(l2)) return kind(l1) < kind(l2);
        switch (kind(l1)) {
            case level_kind::Zero:
            case level_kind::Succ:
                lean_unreachable();  // LCOV_EXCL_LINE
            case level_kind::Param:
            case level_kind::MVar:
                return level_id(l1) < level_id(l2);
            case level_kind::Max:
            case level_kind::IMax:
                if (level_lhs(l1) != level_lhs(l2))
                    return is_norm_lt(level_lhs(l1), level_lhs(l2));
                else
                    return is_norm_lt(level_rhs(l1), level_rhs(l2));
        }
        lean_unreachable();  // LCOV_EXCL_LINE
    } else {
        return p1.second < p2.second;
    }
}

void push_max_args(level const &l, buffer<level> &r) {
    if (is_max(l)) {
        push_max_args(max_lhs(l), r);
        push_max_args(max_rhs(l), r);
    } else {
        r.push_back(l);
```

```
    }
}

level mk_max(buffer<level> const &args) {
    lean_assert(!args.empty());
    unsigned nargs = args.size();
    if (nargs == 1) {
        return args[0];
    } else {
        lean_assert(nargs >= 2);
        level r = mk_max(args[nargs - 2], args[nargs - 1]);
        unsigned i = nargs - 2;
        while (i > 0) {
            --i;
            r = mk_max(args[i], r);
        }
        return r;
    }
}

level mk_succ(level l, unsigned k) {
    while (k > 0) {
        --k;
        l = mk_succ(l);
    }
    return l;
}

level normalize(level const &l) {
    auto p = to_offset(l);
    level const &r = p.first;
    switch (kind(r)) {
        case level_kind::Succ:
            lean_unreachable();  // LCOV_EXCL_LINE
        case level_kind::Zero:
        case level_kind::Param:
        case level_kind::MVar:
            return l;
        case level_kind::IMax: {
            auto l1 = normalize(imax_lhs(r));
            auto l2 = normalize(imax_rhs(r));
            return mk_imax(l1, l2);
        }
        case level_kind::Max: {
            buffer<level> todo;
            buffer<level> args;
            push_max_args(r, todo);
            for (level const &a : todo) push_max_args(normalize(a), args);
            std::sort(args.begin(), args.end(), is_norm_lt);
            buffer<level> &rargs = todo;
            rargs.clear();
            unsigned i = 0;
            if (is_explicit(args[i])) {
                // find max explicit univierse
                while (i + 1 < args.size() && is_explicit(args[i + 1])) i++;
                lean_assert(is_explicit(args[i]));
                unsigned k = to_offset(args[i]).second;
                // an explicit universe k is subsumed by succ^k(l)
                unsigned j = i + 1;
                for (; j < args.size(); j++) {
                    if (to_offset(args[j]).second >= k) break;
                }
                if (j < args.size()) {
                    // explicit universe was subsumed by succ^k'(l) where k' >=
                    // k
                    i++;
                }
            }
            rargs.push_back(args[i]);
            auto p_prev = to_offset(args[i]);
```

```
            i++;
            for (; i < args.size(); i++) {
                auto p_curr = to_offset(args[i]);
                if (p_prev.first == p_curr.first) {
                    if (p_prev.second < p_curr.second) {
                        p_prev = p_curr;
                        rargs.pop_back();
                        rargs.push_back(args[i]);
                    }
                } else {
                    p_prev = p_curr;
                    rargs.push_back(args[i]);
                }
            }
            for (level &a : rargs) a = mk_succ(a, p.second);
            return mk_max(rargs);
        }
    }
    lean_unreachable();  // LCOV_EXCL_LINE
}

bool is_equivalent(level const &lhs, level const &rhs) {
    check_system("level constraints");
    return lhs == rhs || normalize(lhs) == normalize(rhs);
}

bool is_geq_core(level l1, level l2) {
    if (l1 == l2 || is_zero(l2)) return true;
    if (is_max(l2)) return is_geq(l1, max_lhs(l2)) && is_geq(l1, max_rhs(l2));
    if (is_max(l1) && (is_geq(max_lhs(l1), l2) || is_geq(max_rhs(l1), l2)))
        return true;
    if (is_imax(l2))
        return is_geq(l1, imax_lhs(l2)) && is_geq(l1, imax_rhs(l2));
    if (is_imax(l1)) return is_geq(imax_rhs(l1), l2);
    auto p1 = to_offset(l1);
    auto p2 = to_offset(l2);
    if (p1.first == p2.first || is_zero(p2.first))
        return p1.second >= p2.second;
    if (p1.second == p2.second && p1.second > 0)
        return is_geq(p1.first, p2.first);
    return false;
}
bool is_geq(level const &l1, level const &l2) {
    return is_geq_core(normalize(l1), normalize(l2));
}
levels lparams_to_levels(names const &ps) {
    buffer<level> ls;
    for (auto const &p : ps) ls.push_back(mk_univ_param(p));
    return levels(ls);
}

level::level() : level(*g_level_zero) {}

void initialize_level() {
    g_level_zero = new level(lean_level_mk_zero(box(0)));
    mark_persistent(g_level_zero->raw());
    g_level_one = new level(mk_succ(*g_level_zero));
    mark_persistent(g_level_one->raw());
}

void finalize_level() {
    delete g_level_one;
    delete g_level_zero;
}
}  // namespace lean
void print(lean::level const &l) { std::cout << l << std::endl; }
// :::::::::::::
// local_ctx.cpp
// :::::::::::::
/*
```

```
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/sstream.h>

#include <limits>

#include "kernel/abstract.h"
#include "kernel/local_ctx.h"

namespace lean {
static expr *g_dummy_type;
static local_decl *g_dummy_decl;

extern "C" object *lean_mk_local_decl(object *index, object *fvarid,
                                      object *user_name, object *type,
                                      uint8 bi);
extern "C" object *lean_mk_let_decl(object *index, object *fvarid,
                                    object *user_name, object *type,
                                    object *val);
extern "C" uint8 lean_local_decl_binder_info(object *d);

local_decl::local_decl() : object_ref(*g_dummy_decl) {}

local_decl::local_decl(unsigned idx, name const &n, name const &un,
                       expr const &t, expr const &v)
    : object_ref(lean_mk_let_decl(nat(idx).to_obj_arg(), n.to_obj_arg(),
                                  un.to_obj_arg(), t.to_obj_arg(),
                                  v.to_obj_arg())) {}

local_decl::local_decl(unsigned idx, name const &n, name const &un,
                       expr const &t, binder_info bi)
    : object_ref(lean_mk_local_decl(nat(idx).to_obj_arg(), n.to_obj_arg(),
                                    un.to_obj_arg(), t.to_obj_arg(),
                                    static_cast<uint8>(bi))) {}

local_decl::local_decl(local_decl const &d, expr const &t, expr const &v)
    : local_decl(d.get_idx(), d.get_name(), d.get_user_name(), t, v) {}

local_decl::local_decl(local_decl const &d, expr const &t)
    : local_decl(d.get_idx(), d.get_name(), d.get_user_name(), t,
                 d.get_info()) {}

binder_info local_decl::get_info() const {
    return static_cast<binder_info>(lean_local_decl_binder_info(to_obj_arg()));
}

expr local_decl::mk_ref() const { return mk_fvar(get_name()); }

extern "C" object *lean_mk_empty_local_ctx(object *);
extern "C" object *lean_local_ctx_num_indices(object *);
extern "C" uint8 lean_local_ctx_is_empty(object *);
extern "C" object *lean_local_ctx_mk_local_decl(object *lctx, object *name,
                                                object *user_name, object *expr,
                                                uint8 bi);
extern "C" object *lean_local_ctx_mk_let_decl(object *lctx, object *name,
                                              object *user_name, object *type,
                                              object *value);
extern "C" object *lean_local_ctx_find(object *lctx, object *name);
extern "C" object *lean_local_ctx_erase(object *lctx, object *name);

local_ctx::local_ctx() : object_ref(lean_mk_empty_local_ctx(box(0))) {}

bool local_ctx::empty() const { return lean_local_ctx_is_empty(to_obj_arg()); }

local_decl local_ctx::mk_local_decl(name const &n, name const &un,
                                    expr const &type, expr const &value) {
    unsigned idx = unbox(lean_local_ctx_num_indices(to_obj_arg()));
```

```
    m_obj = lean_local_ctx_mk_let_decl(raw(), n.to_obj_arg(), un.to_obj_arg(),
                                       type.to_obj_arg(), value.to_obj_arg());
    return local_decl(idx, n, un, type, value);
}

local_decl local_ctx::mk_local_decl(name const &n, name const &un,
                                    expr const &type, binder_info bi) {
    unsigned idx = unbox(lean_local_ctx_num_indices(to_obj_arg()));
    m_obj =
        lean_local_ctx_mk_local_decl(raw(), n.to_obj_arg(), un.to_obj_arg(),
                                     type.to_obj_arg(), static_cast<uint8>(bi));
    return local_decl(idx, n, un, type, bi);
}

optional<local_decl> local_ctx::find_local_decl(name const &n) const {
    return to_optional<local_decl>(
        lean_local_ctx_find(to_obj_arg(), n.to_obj_arg()));
}

local_decl local_ctx::get_local_decl(name const &n) const {
    if (optional<local_decl> r = find_local_decl(n)) {
        return *r;
    } else {
        // lean_assert(false);
        throw exception(sstream() << "unknown free variable: " << n);
    }
}

expr local_ctx::get_local(name const &n) const {
    lean_assert(find_local_decl(n));
    return get_local_decl(n).mk_ref();
}

void local_ctx::clear(local_decl const &d) {
    m_obj = lean_local_ctx_erase(m_obj, d.get_name().to_obj_arg());
}

template <bool is_lambda>
expr local_ctx::mk_binding(unsigned num, expr const *fvars, expr const &b,
                           bool remove_dead_let) const {
    expr r = abstract(b, num, fvars);
    unsigned i = num;
    while (i > 0) {
        --i;
        local_decl const &decl = get_local_decl(fvar_name(fvars[i]));
        if (optional<expr> const &opt_val = decl.get_value()) {
            if (!remove_dead_let || has_loose_bvar(r, 0)) {
                expr type = abstract(decl.get_type(), i, fvars);
                expr value = abstract(*opt_val, i, fvars);
                r = ::lean::mk_let(decl.get_user_name(), type, value, r);
            } else {
                r = lower_loose_bvars(r, 1, 1);
            }
        } else if (is_lambda) {
            expr type = abstract(decl.get_type(), i, fvars);
            r = ::lean::mk_lambda(decl.get_user_name(), type, r,
                                  decl.get_info());
        } else {
            expr type = abstract(decl.get_type(), i, fvars);
            r = ::lean::mk_pi(decl.get_user_name(), type, r, decl.get_info());
        }
    }
    return r;
}

expr local_ctx::mk_lambda(unsigned num, expr const *fvars, expr const &e,
                          bool remove_dead_let) const {
    return mk_binding<true>(num, fvars, e, remove_dead_let);
}
```

```
expr local_ctx::mk_pi(unsigned num, expr const *fvars, expr const &e,
                      bool remove_dead_let) const {
    return mk_binding<false>(num, fvars, e, remove_dead_let);
}

void initialize_local_ctx() {
    g_dummy_type = new expr(mk_constant(name::mk_internal_unique_name()));
    mark_persistent(g_dummy_type->raw());
    g_dummy_decl = new local_decl(std::numeric_limits<unsigned>::max(),
                                  name("__local_decl_for_default_constructor"),
                                  name("__local_decl_for_default_constructor"),
                                  mk_Prop(), mk_binder_info());
    mark_persistent(g_dummy_decl->raw());
}

void finalize_local_ctx() {
    delete g_dummy_decl;
    delete g_dummy_type;
}
}  // namespace lean
// ::::::::::::::
// quot.cpp
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura

Quotient types.
*/
#include "kernel/local_ctx.h"
#include "kernel/quot.h"
#include "util/name_generator.h"

namespace lean {
name *quot_consts::g_quot = nullptr;
name *quot_consts::g_quot_lift = nullptr;
name *quot_consts::g_quot_ind = nullptr;
name *quot_consts::g_quot_mk = nullptr;

static void check_eq_type(environment const &env) {
    constant_info eq_info = env.get("Eq");
    if (!eq_info.is_inductive())
        throw exception(
            "failed to initialize quot module, environment does not have 'Eq' "
            "type");
    inductive_val eq_val = eq_info.to_inductive_val();
    if (length(eq_info.get_lparams()) != 1)
        throw exception(
            "failed to initialize quot module, unexpected number of universe "
            "params at 'Eq' type");
    if (length(eq_val.get_cnstrs()) != 1)
        throw exception(
            "failed to initialize quot module, unexpected number of "
            "constructors for 'Eq' type");
    local_ctx lctx;
    name_generator g;
    {
        level u = mk_univ_param(head(eq_info.get_lparams()));
        expr alpha =
            lctx.mk_local_decl(g, "α", mk_sort(u), mk_implicit_binder_info());
        expr expected_eq_type =
            lctx.mk_pi(alpha, mk_arrow(alpha, mk_arrow(alpha, mk_Prop())));
        if (expected_eq_type != eq_info.get_type())
            throw exception(
                "failed to initialize quot module, 'Eq' has an expected type");
    }
    {
        constant_info eq_refl_info = env.get(head(eq_val.get_cnstrs()));
```

```
        level u = mk_univ_param(head(eq_refl_info.get_lparams()));
        expr alpha =
            lctx.mk_local_decl(g, "α", mk_sort(u), mk_implicit_binder_info());
        expr a = lctx.mk_local_decl(g, "a", alpha);
        expr expected_eq_refl_type =
            lctx.mk_pi({alpha, a}, mk_app(mk_constant("Eq", {u}), alpha, a, a));
        if (eq_refl_info.get_type() != expected_eq_refl_type)
            throw exception(
                "failed to initialize quot module, unexpected type for 'Eq' "
                "type constructor");
    }
}

environment environment::add_quot() const {
    if (is_quot_initialized()) return *this;
    check_eq_type(*this);
    environment new_env = *this;
    name u_name("u");
    local_ctx lctx;
    name_generator g;
    level u = mk_univ_param(u_name);
    expr Sort_u = mk_sort(u);
    expr alpha = lctx.mk_local_decl(g, "α", Sort_u, mk_implicit_binder_info());
    expr r =
        lctx.mk_local_decl(g, "r", mk_arrow(alpha, mk_arrow(alpha, mk_Prop())));
    /* constant {u} quot {α : Sort u} (r : α → α → Prop) : Sort u */
    new_env.add_core(constant_info(quot_val(*quot_consts::g_quot, {u_name},
                                            lctx.mk_pi({alpha, r}, Sort_u),
                                            quot_kind::Type)));
    expr quot_r = mk_app(mk_constant(*quot_consts::g_quot, {u}), alpha, r);
    expr a = lctx.mk_local_decl(g, "a", alpha);
    /* constant {u} quot.mk {α : Sort u} (r : α → α → Prop) (a : α) : @quot.{u}
     * α r */
    new_env.add_core(constant_info(quot_val(*quot_consts::g_quot_mk, {u_name},
                                            lctx.mk_pi({alpha, r, a}, quot_r),
                                            quot_kind::Mk)));
    /* make r implicit */
    lctx = local_ctx();
    alpha = lctx.mk_local_decl(g, "α", Sort_u, mk_implicit_binder_info());
    r = lctx.mk_local_decl(g, "r", mk_arrow(alpha, mk_arrow(alpha, mk_Prop())),
                           mk_implicit_binder_info());
    quot_r = mk_app(mk_constant(*quot_consts::g_quot, {u}), alpha, r);
    a = lctx.mk_local_decl(g, "a", alpha);
    name v_name("v");
    level v = mk_univ_param(v_name);
    expr Sort_v = mk_sort(v);
    expr beta = lctx.mk_local_decl(g, "β", Sort_v, mk_implicit_binder_info());
    expr f = lctx.mk_local_decl(g, "f", mk_arrow(alpha, beta));
    expr b = lctx.mk_local_decl(g, "b", alpha);
    expr r_a_b = mk_app(r, a, b);
    /* f a = f b */
    expr f_a_eq_f_b =
        mk_app(mk_constant("Eq", {v}), beta, mk_app(f, a), mk_app(f, b));
    /* (∀ a b : α, r a b → f a = f b) */
    expr sanity = lctx.mk_pi({a, b}, mk_arrow(r_a_b, f_a_eq_f_b));
    /* constant {u v} quot.lift {α : Sort u} {r : α → α → Prop} {β : Sort v} (f
       : α → β) : (∀ a b : α, r a b → f a = f b) →  @quot.{u} α r → β */
    new_env.add_core(constant_info(
        quot_val(*quot_consts::g_quot_lift, {u_name, v_name},
                 lctx.mk_pi({alpha, r, beta, f},
                            mk_arrow(sanity, mk_arrow(quot_r, beta))),
                 quot_kind::Lift)));
    /* {β : @quot.{u} α r → Prop} */
    beta = lctx.mk_local_decl(g, "β", mk_arrow(quot_r, mk_Prop()),
                              mk_implicit_binder_info());
    expr quot_mk_a =
        mk_app(mk_constant(*quot_consts::g_quot_mk, {u}), alpha, r, a);
    expr all_quot = lctx.mk_pi(a, mk_app(beta, quot_mk_a));
    expr q = lctx.mk_local_decl(g, "q", quot_r);
    expr beta_q = mk_app(beta, q);
```

```
    /* constant {u} quot.ind {α : Sort u} {r : α → α → Prop} {β : @quot.{u} α r
       → Prop} : (∀ a : α, β (@quot.mk.{u} α r a)) → ∀ q : @quot.{u} α r, β q */
    new_env.add_core(constant_info(quot_val(
        *quot_consts::g_quot_ind, {u_name},
        lctx.mk_pi({alpha, r, beta}, mk_arrow(all_quot, lctx.mk_pi(q, beta_q))),
        quot_kind::Ind)));
    new_env.mark_quot_initialized();
    return new_env;
}

void initialize_quot() {
    quot_consts::g_quot = new name{"Quot"};
    mark_persistent(quot_consts::g_quot->raw());
    quot_consts::g_quot_lift = new name{"Quot", "lift"};
    mark_persistent(quot_consts::g_quot_lift->raw());
    quot_consts::g_quot_ind = new name{"Quot", "ind"};
    mark_persistent(quot_consts::g_quot_ind->raw());
    quot_consts::g_quot_mk = new name{"Quot", "mk"};
    mark_persistent(quot_consts::g_quot_mk->raw());
}

void finalize_quot() {
    delete quot_consts::g_quot;
    delete quot_consts::g_quot_lift;
    delete quot_consts::g_quot_ind;
    delete quot_consts::g_quot_mk;
}
}  // namespace lean
// ::::::::::::::
// replace_fn.cpp
// ::::::::::::::
/*
Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <memory>
#include <vector>

#include "kernel/cache_stack.h"
#include "kernel/replace_fn.h"

#ifndef LEAN_DEFAULT_REPLACE_CACHE_CAPACITY
#define LEAN_DEFAULT_REPLACE_CACHE_CAPACITY 1024 * 8
#endif

namespace lean {
struct replace_cache {
    struct entry {
        object *m_cell;
        unsigned m_offset;
        expr m_result;
        entry() : m_cell(nullptr) {}
    };
    unsigned m_capacity;
    std::vector<entry> m_cache;
    std::vector<unsigned> m_used;
    replace_cache(unsigned c) : m_capacity(c), m_cache(c) {}

    expr *find(expr const &e, unsigned offset) {
        unsigned i = hash(hash(e), offset) % m_capacity;
        if (m_cache[i].m_cell == e.raw() && m_cache[i].m_offset == offset)
            return &m_cache[i].m_result;
        else
            return nullptr;
    }

    void insert(expr const &e, unsigned offset, expr const &v) {
        unsigned i = hash(hash(e), offset) % m_capacity;
```

```
        if (m_cache[i].m_cell == nullptr) m_used.push_back(i);
        m_cache[i].m_cell = e.raw();
        m_cache[i].m_offset = offset;
        m_cache[i].m_result = v;
    }

    void clear() {
        for (unsigned i : m_used) {
            m_cache[i].m_cell = nullptr;
            m_cache[i].m_result = expr();
        }
        m_used.clear();
    }
};

/* CACHE_RESET: NO */
MK_CACHE_STACK(replace_cache, LEAN_DEFAULT_REPLACE_CACHE_CAPACITY)

class replace_rec_fn {
    replace_cache_ref m_cache;
    std::function<optional<expr>(expr const &, unsigned)> m_f;
    bool m_use_cache;

    expr save_result(expr const &e, unsigned offset, expr const &r,
                     bool shared) {
        if (shared) m_cache->insert(e, offset, r);
        return r;
    }

    expr apply(expr const &e, unsigned offset) {
        bool shared = false;
        if (m_use_cache && is_shared(e)) {
            if (auto r = m_cache->find(e, offset)) return *r;
            shared = true;
        }
        check_system("replace");

        if (optional<expr> r = m_f(e, offset)) {
            return save_result(e, offset, *r, shared);
        } else {
            switch (e.kind()) {
                case expr_kind::Const:
                case expr_kind::Sort:
                case expr_kind::BVar:
                case expr_kind::Lit:
                case expr_kind::MVar:
                case expr_kind::FVar:
                    return save_result(e, offset, e, shared);
                case expr_kind::MData: {
                    expr new_e = apply(mdata_expr(e), offset);
                    return save_result(e, offset, update_mdata(e, new_e),
                                       shared);
                }
                case expr_kind::Proj: {
                    expr new_e = apply(proj_expr(e), offset);
                    return save_result(e, offset, update_proj(e, new_e),
                                       shared);
                }
                case expr_kind::App: {
                    expr new_f = apply(app_fn(e), offset);
                    expr new_a = apply(app_arg(e), offset);
                    return save_result(e, offset, update_app(e, new_f, new_a),
                                       shared);
                }
                case expr_kind::Pi:
                case expr_kind::Lambda: {
                    expr new_d = apply(binding_domain(e), offset);
                    expr new_b = apply(binding_body(e), offset + 1);
                    return save_result(e, offset,
                                       update_binding(e, new_d, new_b), shared);
```

```
                }
                case expr_kind::Let: {
                    expr new_t = apply(let_type(e), offset);
                    expr new_v = apply(let_value(e), offset);
                    expr new_b = apply(let_body(e), offset + 1);
                    return save_result(
                        e, offset, update_let(e, new_t, new_v, new_b), shared);
                }
            }
            lean_unreachable();
        }
    }

   public:
    template <typename F>
    replace_rec_fn(F const &f, bool use_cache)
        : m_f(f), m_use_cache(use_cache) {}

    expr operator()(expr const &e) { return apply(e, 0); }
};

expr replace(expr const &e,
             std::function<optional<expr>(expr const &, unsigned)> const &f,
             bool use_cache) {
    return replace_rec_fn(f, use_cache)(e);
}
}  // namespace lean
// :::::::::::::
// type_checker.cpp
// :::::::::::::
/*
Copyright (c) 2013-14 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <lean/flet.h>
#include <lean/interrupt.h>
#include <lean/sstream.h>

#include <utility>
#include <vector>

#include "kernel/abstract.h"
#include "kernel/expr_maps.h"
#include "kernel/for_each_fn.h"
#include "kernel/inductive.h"
#include "kernel/instantiate.h"
#include "kernel/kernel_exception.h"
#include "kernel/quot.h"
#include "kernel/replace_fn.h"
#include "kernel/type_checker.h"
#include "util/lbool.h"

namespace lean {
static name *g_kernel_fresh = nullptr;
static expr *g_dont_care = nullptr;
static expr *g_nat_zero = nullptr;
static expr *g_nat_succ = nullptr;
static expr *g_nat_add = nullptr;
static expr *g_nat_sub = nullptr;
static expr *g_nat_mul = nullptr;
static expr *g_nat_mod = nullptr;
static expr *g_nat_div = nullptr;
static expr *g_nat_beq = nullptr;
static expr *g_nat_ble = nullptr;

type_checker::state::state(environment const &env)
    : m_env(env), m_ngen(*g_kernel_fresh) {}

```

```cpp
/** \brief Make sure \c e "is" a sort, and return the corresponding sort.
    If \c e is not a sort, then the whnf procedure is invoked.

    \remark \c s is used to extract position (line number information) when an
    error message is produced */
expr type_checker::ensure_sort_core(expr e, expr const &s) {
    if (is_sort(e)) return e;
    auto new_e = whnf(e);
    if (is_sort(new_e)) {
        return new_e;
    } else {
        throw type_expected_exception(env(), m_lctx, s);
    }
}

/** \brief Similar to \c ensure_sort, but makes sure \c e "is" a Pi. */
expr type_checker::ensure_pi_core(expr e, expr const &s) {
    if (is_pi(e)) return e;
    auto new_e = whnf(e);
    if (is_pi(new_e)) {
        return new_e;
    } else {
        throw function_expected_exception(env(), m_lctx, s);
    }
}

void type_checker::check_level(level const &l) {
    if (m_lparams) {
        if (auto n2 = get_undef_param(l, *m_lparams))
            throw kernel_exception(
                env(), sstream() << "invalid reference to undefined universe "
                                    "level parameter '"
                                 << *n2 << "'");
    }
}

expr type_checker::infer_fvar(expr const &e) {
    if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
        return decl->get_type();
    } else {
        throw kernel_exception(env(), "unknown free variable");
    }
}

expr type_checker::infer_constant(expr const &e, bool infer_only) {
    constant_info info = env().get(const_name(e));
    auto const &ps = info.get_lparams();
    auto const &ls = const_levels(e);
    if (length(ps) != length(ls))
        throw kernel_exception(
            env(), sstream()
                       << "incorrect number of universe levels parameters for '"
                       << const_name(e) << "', #" << length(ps)
                       << " expected, #" << length(ls) << " provided");
    if (!infer_only) {
        if (m_safe_only && info.is_unsafe()) {
            throw kernel_exception(
                env(),
                sstream() << "invalid declaration, it uses unsafe declaration '"
                          << const_name(e) << "'");
        }
        for (level const &l : ls) check_level(l);
    }
    return instantiate_type_lparams(info, ls);
}

expr type_checker::infer_lambda(expr const &_e, bool infer_only) {
    flet<local_ctx> save_lctx(m_lctx, m_lctx);
    buffer<expr> fvars;
    expr e = _e;
```

```
    while (is_lambda(e)) {
        expr d = instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
        expr fvar = m_lctx.mk_local_decl(m_st->m_ngen, binding_name(e), d,
                                         binding_info(e));
        fvars.push_back(fvar);
        if (!infer_only) {
            ensure_sort_core(infer_type_core(d, infer_only), d);
        }
        e = binding_body(e);
    }
    expr r = infer_type_core(instantiate_rev(e, fvars.size(), fvars.data()),
                             infer_only);
    r = cheap_beta_reduce(r);
    return m_lctx.mk_pi(fvars, r);
}

expr type_checker::infer_pi(expr const &_e, bool infer_only) {
    flet<local_ctx> save_lctx(m_lctx, m_lctx);
    buffer<expr> fvars;
    buffer<level> us;
    expr e = _e;
    while (is_pi(e)) {
        expr d = instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
        expr t1 = ensure_sort_core(infer_type_core(d, infer_only), d);
        us.push_back(sort_level(t1));
        expr fvar = m_lctx.mk_local_decl(m_st->m_ngen, binding_name(e), d,
                                         binding_info(e));
        fvars.push_back(fvar);
        e = binding_body(e);
    }
    e = instantiate_rev(e, fvars.size(), fvars.data());
    expr s = ensure_sort_core(infer_type_core(e, infer_only), e);
    level r = sort_level(s);
    unsigned i = fvars.size();
    while (i > 0) {
        --i;
        r = mk_imax(us[i], r);
    }
    return mk_sort(r);
}

expr type_checker::infer_app(expr const &e, bool infer_only) {
    if (!infer_only) {
        expr f_type = ensure_pi_core(infer_type_core(app_fn(e), infer_only), e);
        expr a_type = infer_type_core(app_arg(e), infer_only);
        expr d_type = binding_domain(f_type);
        if (!is_def_eq(a_type, d_type)) {
            throw app_type_mismatch_exception(env(), m_lctx, e, f_type, a_type);
        }
        return instantiate(binding_body(f_type), app_arg(e));
    } else {
        buffer<expr> args;
        expr const &f = get_app_args(e, args);
        expr f_type = infer_type_core(f, true);
        unsigned j = 0;
        unsigned nargs = args.size();
        for (unsigned i = 0; i < nargs; i++) {
            if (is_pi(f_type)) {
                f_type = binding_body(f_type);
            } else {
                f_type = instantiate_rev(f_type, i - j, args.data() + j);
                f_type = ensure_pi_core(f_type, e);
                f_type = binding_body(f_type);
                j = i;
            }
        }
        return instantiate_rev(f_type, nargs - j, args.data() + j);
    }
}
```

```
static void mark_used(unsigned n, expr const *fvars, expr const &b,
                      bool *used) {
    if (!has_fvar(b)) return;
    for_each(b, [&](expr const &x, unsigned) {
        if (!has_fvar(x)) return false;
        if (is_fvar(x)) {
            for (unsigned i = 0; i < n; i++) {
                if (fvar_name(fvars[i]) == fvar_name(x)) {
                    used[i] = true;
                    return false;
                }
            }
        }
        return true;
    });
}

expr type_checker::infer_let(expr const &_e, bool infer_only) {
    flet<local_ctx> save_lctx(m_lctx, m_lctx);
    buffer<expr> fvars;
    buffer<expr> vals;
    expr e = _e;
    while (is_let(e)) {
        expr type = instantiate_rev(let_type(e), fvars.size(), fvars.data());
        expr val = instantiate_rev(let_value(e), fvars.size(), fvars.data());
        expr fvar = m_lctx.mk_local_decl(m_st->m_ngen, let_name(e), type, val);
        fvars.push_back(fvar);
        vals.push_back(val);
        if (!infer_only) {
            ensure_sort_core(infer_type_core(type, infer_only), type);
            expr val_type = infer_type_core(val, infer_only);
            if (!is_def_eq(val_type, type)) {
                throw def_type_mismatch_exception(env(), m_lctx, let_name(e),
                                                  val_type, type);
            }
        }
        e = let_body(e);
    }
    expr r = infer_type_core(instantiate_rev(e, fvars.size(), fvars.data()),
                             infer_only);
    r = cheap_beta_reduce(r);  // use `cheap_beta_reduce` (to try) to reduce
                               // number of dependencies
    buffer<bool, 128> used;
    used.resize(fvars.size(), false);
    mark_used(fvars.size(), fvars.data(), r, used.data());
    unsigned i = fvars.size();
    while (i > 0) {
        --i;
        if (used[i]) mark_used(i, fvars.data(), vals[i], used.data());
    }
    buffer<expr> used_fvars;
    for (unsigned i = 0; i < fvars.size(); i++) {
        if (used[i]) used_fvars.push_back(fvars[i]);
    }
    return m_lctx.mk_pi(used_fvars, r);
}

expr type_checker::infer_proj(expr const &e, bool infer_only) {
    expr type = whnf(infer_type_core(proj_expr(e), infer_only));
    if (!proj_idx(e).is_small()) throw invalid_proj_exception(env(), m_lctx, e);
    unsigned idx = proj_idx(e).get_small_value();
    buffer<expr> args;
    expr const &I = get_app_args(type, args);
    if (!is_constant(I)) throw invalid_proj_exception(env(), m_lctx, e);
    name const &I_name = const_name(I);
    if (I_name != proj_sname(e)) throw invalid_proj_exception(env(), m_lctx, e);
    constant_info I_info = env().get(I_name);
    if (!I_info.is_inductive()) throw invalid_proj_exception(env(), m_lctx, e);
    inductive_val I_val = I_info.to_inductive_val();
    if (length(I_val.get_cnstrs()) != 1 ||
```

```
        args.size() != I_val.get_nparams() + I_val.get_nindices())
        throw invalid_proj_exception(env(), m_lctx, e);

    constant_info c_info = env().get(head(I_val.get_cnstrs()));
    expr r = instantiate_type_lparams(c_info, const_levels(I));
    for (unsigned i = 0; i < I_val.get_nparams(); i++) {
        lean_assert(i < args.size());
        r = whnf(r);
        if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
        r = instantiate(binding_body(r), args[i]);
    }
    for (unsigned i = 0; i < idx; i++) {
        r = whnf(r);
        if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
        if (has_loose_bvars(binding_body(r)))
            r = instantiate(binding_body(r), mk_proj(I_name, i, proj_expr(e)));
        else
            r = binding_body(r);
    }
    r = whnf(r);
    if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
    return binding_domain(r);
}

/** \brief Return type of expression \c e, if \c infer_only is false, then it
   also check whether \c e is type correct or not. \pre closed(e) */
expr type_checker::infer_type_core(expr const &e, bool infer_only) {
    if (is_bvar(e))
        throw kernel_exception(
            env(),
            "type checker does not support loose bound variables, replace them "
            "with free variables before invoking it");

    lean_assert(!has_loose_bvars(e));
    check_system("type checker");

    auto it = m_st->m_infer_type[infer_only].find(e);
    if (it != m_st->m_infer_type[infer_only].end()) return it->second;

    expr r;
    switch (e.kind()) {
        case expr_kind::Lit:
            r = lit_type(lit_value(e));
            break;
        case expr_kind::MData:
            r = infer_type_core(mdata_expr(e), infer_only);
            break;
        case expr_kind::Proj:
            r = infer_proj(e, infer_only);
            break;
        case expr_kind::FVar:
            r = infer_fvar(e);
            break;
        case expr_kind::MVar:
            throw kernel_exception(
                env(), "kernel type checker does not support meta variables");
        case expr_kind::BVar:
            lean_unreachable();  // LCOV_EXCL_LINE
        case expr_kind::Sort:
            if (!infer_only) check_level(sort_level(e));
            r = mk_sort(mk_succ(sort_level(e)));
            break;
        case expr_kind::Const:
            r = infer_constant(e, infer_only);
            break;
        case expr_kind::Lambda:
            r = infer_lambda(e, infer_only);
            break;
        case expr_kind::Pi:
            r = infer_pi(e, infer_only);
```

```
            break;
        case expr_kind::App:
            r = infer_app(e, infer_only);
            break;
        case expr_kind::Let:
            r = infer_let(e, infer_only);
            break;
    }

    m_st->m_infer_type[infer_only].insert(mk_pair(e, r));
    return r;
}

expr type_checker::infer_type(expr const &e) {
    return infer_type_core(e, true);
}

expr type_checker::check(expr const &e, names const &lps) {
    flet<names const *> updt(m_lparams, &lps);
    return infer_type_core(e, false);
}

expr type_checker::check_ignore_undefined_universes(expr const &e) {
    flet<names const *> updt(m_lparams, nullptr);
    return infer_type_core(e, false);
}

expr type_checker::ensure_sort(expr const &e, expr const &s) {
    return ensure_sort_core(e, s);
}

expr type_checker::ensure_pi(expr const &e, expr const &s) {
    return ensure_pi_core(e, s);
}

/** \brief Return true iff \c e is a proposition */
bool type_checker::is_prop(expr const &e) {
    return whnf(infer_type(e)) == mk_Prop();
}

/** \brief Apply normalizer extensions to \c e.
    If `cheap == true`, then we don't perform delta-reduction when reducing
   major premise. */
optional<expr> type_checker::reduce_recursor(expr const &e, bool cheap) {
    if (env().is_quot_initialized()) {
        if (optional<expr> r =
                quot_reduce_rec(e, [&](expr const &e) { return whnf(e); })) {
            return r;
        }
    }
    if (optional<expr> r = inductive_reduce_rec(
            env(), e,
            [&](expr const &e) {
                return cheap ? whnf_core(e, cheap) : whnf(e);
            },
            [&](expr const &e) { return infer(e); },
            [&](expr const &e1, expr const &e2) {
                return is_def_eq(e1, e2);
            })) {
        return r;
    }
    return none_expr();
}

expr type_checker::whnf_fvar(expr const &e, bool cheap) {
    if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
        if (optional<expr> const &v = decl->get_value()) {
            /* zeta-reduction */
            return whnf_core(*v, cheap);
        }
```

```
    }
    return e;
}

/* If `cheap == true`, then we don't perform delta-reduction when reducing major
 * premise. */
optional<expr> type_checker::reduce_proj(expr const &e, bool cheap) {
    if (!proj_idx(e).is_small()) return none_expr();
    unsigned idx = proj_idx(e).get_small_value();
    expr c;
    if (cheap)
        c = whnf_core(proj_expr(e), cheap);
    else
        c = whnf(proj_expr(e));
    buffer<expr> args;
    expr const &mk = get_app_args(c, args);
    if (!is_constant(mk)) return none_expr();
    constant_info mk_info = env().get(const_name(mk));
    if (!mk_info.is_constructor()) return none_expr();
    unsigned nparams = mk_info.to_constructor_val().get_nparams();
    if (nparams + idx < args.size())
        return some_expr(args[nparams + idx]);
    else
        return none_expr();
}

static bool is_let_fvar(local_ctx const &lctx, expr const &e) {
    lean_assert(is_fvar(e));
    if (optional<local_decl> decl = lctx.find_local_decl(e)) {
        return static_cast<bool>(decl->get_value());
    } else {
        return false;
    }
}

/** \brief Weak head normal form core procedure. It does not perform delta
   reduction nor normalization extensions. If `cheap == true`, then we don't
   perform delta-reduction when reducing major premise of recursors and
   projections. We also do not cache results. */
expr type_checker::whnf_core(expr const &e, bool cheap) {
    check_system("whnf");

    // handle easy cases
    switch (e.kind()) {
        case expr_kind::BVar:
        case expr_kind::Sort:
        case expr_kind::MVar:
        case expr_kind::Pi:
        case expr_kind::Const:
        case expr_kind::Lambda:
        case expr_kind::Lit:
            return e;
        case expr_kind::MData:
            return whnf_core(mdata_expr(e), cheap);
        case expr_kind::FVar:
            if (is_let_fvar(m_lctx, e))
                break;
            else
                return e;
        case expr_kind::App:
        case expr_kind::Let:
        case expr_kind::Proj:
            break;
    }

    // check cache
    if (!cheap) {
        auto it = m_st->m_whnf_core.find(e);
        if (it != m_st->m_whnf_core.end()) return it->second;
    }
```

```

    // do the actual work
    expr r;
    switch (e.kind()) {
        case expr_kind::BVar:
        case expr_kind::Sort:
        case expr_kind::MVar:
        case expr_kind::Pi:
        case expr_kind::Const:
        case expr_kind::Lambda:
        case expr_kind::Lit:
        case expr_kind::MData:
            lean_unreachable();  // LCOV_EXCL_LINE
        case expr_kind::FVar:
            return whnf_fvar(e, cheap);
        case expr_kind::Proj: {
            if (auto m = reduce_proj(e, cheap))
                r = whnf_core(*m, cheap);
            else
                r = e;
            break;
        }
        case expr_kind::App: {
            buffer<expr> args;
            expr f0 = get_app_rev_args(e, args);
            expr f = whnf_core(f0, cheap);
            if (is_lambda(f)) {
                unsigned m = 1;
                unsigned num_args = args.size();
                while (is_lambda(binding_body(f)) && m < num_args) {
                    f = binding_body(f);
                    m++;
                }
                lean_assert(m <= num_args);
                r = whnf_core(
                    mk_rev_app(instantiate(binding_body(f), m,
                                           args.data() + (num_args - m)),
                               num_args - m, args.data()),
                    cheap);
            } else if (f == f0) {
                if (auto r = reduce_recursor(e, cheap)) {
                    /* iota-reduction and quotient reduction rules */
                    return whnf_core(*r, cheap);
                } else {
                    return e;
                }
            } else {
                r = whnf_core(mk_rev_app(f, args.size(), args.data()), cheap);
            }
            break;
        }
        case expr_kind::Let:
            r = whnf_core(instantiate(let_body(e), let_value(e)), cheap);
            break;
    }

    if (!cheap) {
        m_st->m_whnf_core.insert(mk_pair(e, r));
    }
    return r;
}

/** \brief Return some definition \c d iff \c e is a target for delta-reduction,
   and the given definition is the one to be expanded. */
optional<constant_info> type_checker::is_delta(expr const &e) const {
    expr const &f = get_app_fn(e);
    if (is_constant(f)) {
        if (optional<constant_info> info = env().find(const_name(f)))
            if (info->has_value()) return info;
    }
```

```
    return none_constant_info();
}

optional<expr> type_checker::unfold_definition_core(expr const &e) {
    if (is_constant(e)) {
        if (auto d = is_delta(e)) {
            if (length(const_levels(e)) == d->get_num_lparams())
                return some_expr(
                    instantiate_value_lparams(*d, const_levels(e)));
        }
    }
    return none_expr();
}

/* Unfold head(e) if it is a constant */
optional<expr> type_checker::unfold_definition(expr const &e) {
    if (is_app(e)) {
        expr f0 = get_app_fn(e);
        if (auto f = unfold_definition_core(f0)) {
            buffer<expr> args;
            get_app_rev_args(e, args);
            return some_expr(mk_rev_app(*f, args));
        } else {
            return none_expr();
        }
    } else {
        return unfold_definition_core(e);
    }
}

static expr *g_lean_reduce_bool = nullptr;
static expr *g_lean_reduce_nat = nullptr;

namespace ir {
object *run_boxed(environment const &env, options const &opts, name const &fn,
                  unsigned n, object **args);
}

expr mk_bool_true();
expr mk_bool_false();

optional<expr> reduce_native(environment const &env, expr const &e) {
    if (!is_app(e)) return none_expr();
    expr const &arg = app_arg(e);
    if (!is_constant(arg)) return none_expr();
    if (app_fn(e) == *g_lean_reduce_bool) {
        object *r = ir::run_boxed(env, options(), const_name(arg), 0, nullptr);
        if (!lean_is_scalar(r)) {
            lean_dec_ref(r);
            throw kernel_exception(env,
                                   "type checker failure, unexpected result "
                                   "value for 'Lean.reduceBool'");
        }
        return lean_unbox(r) == 0 ? some_expr(mk_bool_false())
                                  : some_expr(mk_bool_true());
    }
    if (app_fn(e) == *g_lean_reduce_nat) {
        object *r = ir::run_boxed(env, options(), const_name(arg), 0, nullptr);
        if (lean_is_scalar(r) || lean_is_mpz(r)) {
            return some_expr(mk_lit(literal(nat(r))));
        } else {
            throw kernel_exception(env,
                                   "type checker failure, unexpected result "
                                   "value for 'Lean.reduceNat'");
        }
    }
    return none_expr();
}

static inline bool is_nat_lit_ext(expr const &e) {
```

```
    return e == *g_nat_zero || is_nat_lit(e);
}
static inline nat get_nat_val(expr const &e) {
    lean_assert(is_nat_lit_ext(e));
    if (e == *g_nat_zero) return nat((unsigned)0);
    return lit_value(e).get_nat();
}

template <typename F>
optional<expr> type_checker::reduce_bin_nat_op(F const &f, expr const &e) {
    expr arg1 = whnf(app_arg(app_fn(e)));
    if (!is_nat_lit_ext(arg1)) return none_expr();
    expr arg2 = whnf(app_arg(e));
    if (!is_nat_lit_ext(arg2)) return none_expr();
    nat v1 = get_nat_val(arg1);
    nat v2 = get_nat_val(arg2);
    return some_expr(mk_lit(literal(nat(f(v1.raw(), v2.raw())))));
}

template <typename F>
optional<expr> type_checker::reduce_bin_nat_pred(F const &f, expr const &e) {
    expr arg1 = whnf(app_arg(app_fn(e)));
    if (!is_nat_lit_ext(arg1)) return none_expr();
    expr arg2 = whnf(app_arg(e));
    if (!is_nat_lit_ext(arg2)) return none_expr();
    nat v1 = get_nat_val(arg1);
    nat v2 = get_nat_val(arg2);
    return f(v1.raw(), v2.raw()) ? some_expr(mk_bool_true())
                                 : some_expr(mk_bool_false());
}

optional<expr> type_checker::reduce_nat(expr const &e) {
    if (has_fvar(e)) return none_expr();
    unsigned nargs = get_app_num_args(e);
    if (nargs == 1) {
        expr const &f = app_fn(e);
        if (f == *g_nat_succ) {
            expr arg = whnf(app_arg(e));
            if (!is_nat_lit_ext(arg)) return none_expr();
            nat v = get_nat_val(arg);
            return some_expr(mk_lit(literal(nat(v + nat(1)))));
        }
    } else if (nargs == 2) {
        expr const &f = app_fn(app_fn(e));
        if (!is_constant(f)) return none_expr();
        if (f == *g_nat_add) return reduce_bin_nat_op(nat_add, e);
        if (f == *g_nat_sub) return reduce_bin_nat_op(nat_sub, e);
        if (f == *g_nat_mul) return reduce_bin_nat_op(nat_mul, e);
        if (f == *g_nat_mod) return reduce_bin_nat_op(nat_mod, e);
        if (f == *g_nat_div) return reduce_bin_nat_op(nat_div, e);
        if (f == *g_nat_beq) return reduce_bin_nat_pred(nat_eq, e);
        if (f == *g_nat_ble) return reduce_bin_nat_pred(nat_le, e);
    }
    return none_expr();
}

/** \brief Put expression \c t in weak head normal form */
expr type_checker::whnf(expr const &e) {
    // Do not cache easy cases
    switch (e.kind()) {
        case expr_kind::BVar:
        case expr_kind::Sort:
        case expr_kind::MVar:
        case expr_kind::Pi:
        case expr_kind::Lit:
            return e;
        case expr_kind::MData:
            return whnf(mdata_expr(e));
        case expr_kind::FVar:
            if (is_let_fvar(m_lctx, e))
```

```
                break;
            else
                return e;
        case expr_kind::Lambda:
        case expr_kind::App:
        case expr_kind::Const:
        case expr_kind::Let:
        case expr_kind::Proj:
            break;
    }

    // check cache
    auto it = m_st->m_whnf.find(e);
    if (it != m_st->m_whnf.end()) return it->second;

    expr t = e;
    while (true) {
        expr t1 = whnf_core(t);
        if (auto v = reduce_native(env(), t1)) {
            m_st->m_whnf.insert(mk_pair(e, *v));
            return *v;
        } else if (auto v = reduce_nat(t1)) {
            m_st->m_whnf.insert(mk_pair(e, *v));
            return *v;
        } else if (auto next_t = unfold_definition(t1)) {
            t = *next_t;
        } else {
            auto r = t1;
            m_st->m_whnf.insert(mk_pair(e, r));
            return r;
        }
    }
}

/** \brief Given lambda/Pi expressions \c t and \c s, return true iff \c t is
   def eq to \c s.

        t and s are definitionally equal
           iff
        domain(t) is definitionally equal to domain(s)
        and
        body(t) is definitionally equal to body(s) */
bool type_checker::is_def_eq_binding(expr t, expr s) {
    lean_assert(t.kind() == s.kind());
    lean_assert(is_binding(t));
    flet<local_ctx> save_lctx(m_lctx, m_lctx);
    expr_kind k = t.kind();
    buffer<expr> subst;
    do {
        optional<expr> var_s_type;
        if (binding_domain(t) != binding_domain(s)) {
            var_s_type =
                instantiate_rev(binding_domain(s), subst.size(), subst.data());
            expr var_t_type =
                instantiate_rev(binding_domain(t), subst.size(), subst.data());
            if (!is_def_eq(var_t_type, *var_s_type)) return false;
        }
        if (has_loose_bvars(binding_body(t)) ||
            has_loose_bvars(binding_body(s))) {
            // free variable is used inside t or s
            if (!var_s_type)
                var_s_type = instantiate_rev(binding_domain(s), subst.size(),
                                             subst.data());
            subst.push_back(m_lctx.mk_local_decl(m_st->m_ngen, binding_name(s),
                                                 *var_s_type, binding_info(s)));
        } else {
            subst.push_back(*g_dont_care);  // don't care
        }
        t = binding_body(t);
        s = binding_body(s);
```

```cpp
    } while (t.kind() == k && s.kind() == k);
    return is_def_eq(instantiate_rev(t, subst.size(), subst.data()),
                     instantiate_rev(s, subst.size(), subst.data()));
}

bool type_checker::is_def_eq(level const &l1, level const &l2) {
    if (is_equivalent(l1, l2)) {
        return true;
    } else {
        return false;
    }
}

bool type_checker::is_def_eq(levels const &ls1, levels const &ls2) {
    if (is_nil(ls1) && is_nil(ls2)) {
        return true;
    } else if (!is_nil(ls1) && !is_nil(ls2)) {
        return is_def_eq(head(ls1), head(ls2)) &&
               is_def_eq(tail(ls1), tail(ls2));
    } else {
        return false;
    }
}

/** \brief This is an auxiliary method for is_def_eq. It handles the "easy
 * cases". */
lbool type_checker::quick_is_def_eq(expr const &t, expr const &s,
                                    bool use_hash) {
    if (m_st->m_eqv_manager.is_equiv(t, s, use_hash)) return l_true;
    if (t.kind() == s.kind()) {
        switch (t.kind()) {
            case expr_kind::Lambda:
            case expr_kind::Pi:
                return to_lbool(is_def_eq_binding(t, s));
            case expr_kind::Sort:
                return to_lbool(is_def_eq(sort_level(t), sort_level(s)));
            case expr_kind::MData:
                return to_lbool(is_def_eq(mdata_expr(t), mdata_expr(s)));
            case expr_kind::MVar:
                lean_unreachable();  // LCOV_EXCL_LINE
            case expr_kind::BVar:
            case expr_kind::FVar:
            case expr_kind::App:
            case expr_kind::Const:
            case expr_kind::Let:
            case expr_kind::Proj:
                // We do not handle these cases in this method.
                break;
            case expr_kind::Lit:
                return to_lbool(lit_value(t) == lit_value(s));
        }
    }
    return l_undef;  // This is not an "easy case"
}

/** \brief Return true if arguments of \c t are definitionally equal to
   arguments of \c s. This method is used to implement an optimization in the
   method \c is_def_eq. */
bool type_checker::is_def_eq_args(expr t, expr s) {
    while (is_app(t) && is_app(s)) {
        if (!is_def_eq(app_arg(t), app_arg(s))) return false;
        t = app_fn(t);
        s = app_fn(s);
    }
    return !is_app(t) && !is_app(s);
}

/** \brief Try to solve (fun (x : A), B) =?= s by trying eta-expansion on s */
bool type_checker::try_eta_expansion_core(expr const &t, expr const &s) {
    if (is_lambda(t) && !is_lambda(s)) {
```

```
        expr s_type = whnf(infer_type(s));
        if (!is_pi(s_type)) return false;
        expr new_s = mk_lambda(binding_name(s_type), binding_domain(s_type),
                               mk_app(s, mk_bvar(0)), binding_info(s_type));
        if (!is_def_eq(t, new_s)) return false;
        return true;
    } else {
        return false;
    }
}

/** \brief Return true if \c t and \c s are definitionally equal because they
   are applications of the form <tt>(f a_1 ... a_n)</tt> <tt>(g b_1 ...
   b_n)</tt>, and \c f and \c g are definitionally equal, and \c a_i and \c b_i
   are also definitionally equal for every 1 <= i <= n.
    Return false otherwise. */
bool type_checker::is_def_eq_app(expr const &t, expr const &s) {
    if (is_app(t) && is_app(s)) {
        buffer<expr> t_args;
        buffer<expr> s_args;
        expr t_fn = get_app_args(t, t_args);
        expr s_fn = get_app_args(s, s_args);
        if (is_def_eq(t_fn, s_fn) && t_args.size() == s_args.size()) {
            unsigned i = 0;
            for (; i < t_args.size(); i++) {
                if (!is_def_eq(t_args[i], s_args[i])) break;
            }
            if (i == t_args.size()) return true;
        }
    }
    return false;
}

/** \brief Return true if \c t and \c s are definitionally equal due to proof
   irrelevant. Return false otherwise. */
bool type_checker::is_def_eq_proof_irrel(expr const &t, expr const &s) {
    // Proof irrelevance support for Prop (aka Type.{0})
    expr t_type = infer_type(t);
    if (!is_prop(t_type)) return false;
    expr s_type = infer_type(s);
    return is_def_eq(t_type, s_type);
}

bool type_checker::failed_before(expr const &t, expr const &s) const {
    if (hash(t) < hash(s)) {
        return m_st->m_failure.find(mk_pair(t, s)) != m_st->m_failure.end();
    } else if (hash(t) > hash(s)) {
        return m_st->m_failure.find(mk_pair(s, t)) != m_st->m_failure.end();
    } else {
        return m_st->m_failure.find(mk_pair(t, s)) != m_st->m_failure.end() ||
               m_st->m_failure.find(mk_pair(s, t)) != m_st->m_failure.end();
    }
}

void type_checker::cache_failure(expr const &t, expr const &s) {
    if (hash(t) <= hash(s))
        m_st->m_failure.insert(mk_pair(t, s));
    else
        m_st->m_failure.insert(mk_pair(s, t));
}

/** \brief Perform one lazy delta-reduction step.
     Return
     - l_true if t_n and s_n are definitionally equal.
     - l_false if they are not definitionally equal.
     - l_undef it the step did not manage to establish whether they are
   definitionally equal or not.

     \remark t_n, s_n and cs are updated. */
auto type_checker::lazy_delta_reduction_step(expr &t_n, expr &s_n)
```

```
    -> reduction_status {
    auto d_t = is_delta(t_n);
    auto d_s = is_delta(s_n);
    if (!d_t && !d_s) {
        return reduction_status::DefUnknown;
    } else if (d_t && !d_s) {
        t_n = whnf_core(*unfold_definition(t_n));
    } else if (!d_t && d_s) {
        s_n = whnf_core(*unfold_definition(s_n));
    } else {
        int c = compare(d_t->get_hints(), d_s->get_hints());
        if (c < 0) {
            t_n = whnf_core(*unfold_definition(t_n));
        } else if (c > 0) {
            s_n = whnf_core(*unfold_definition(s_n));
        } else {
            if (is_app(t_n) && is_app(s_n) && is_eqp(*d_t, *d_s)) {
                // Optimization:
                // We try to check if their arguments are definitionally equal.
                // If they are, then t_n and s_n must be definitionally equal,
                // and we can skip the delta-reduction step.
                if (!failed_before(t_n, s_n)) {
                    if (is_def_eq(const_levels(get_app_fn(t_n)),
                                  const_levels(get_app_fn(s_n))) &&
                        is_def_eq_args(t_n, s_n)) {
                        return reduction_status::DefEqual;
                    } else {
                        cache_failure(t_n, s_n);
                    }
                }
            }
            t_n = whnf_core(*unfold_definition(t_n));
            s_n = whnf_core(*unfold_definition(s_n));
        }
    }
    switch (quick_is_def_eq(t_n, s_n)) {
        case l_true:
            return reduction_status::DefEqual;
        case l_false:
            return reduction_status::DefDiff;
        case l_undef:
            return reduction_status::Continue;
    }
    lean_unreachable();
}

inline bool is_nat_zero(expr const &t) {
    return t == *g_nat_zero || (is_nat_lit(t) && lit_value(t).is_zero());
}

inline optional<expr> is_nat_succ(expr const &t) {
    if (is_nat_lit(t)) {
        nat val = lit_value(t).get_nat();
        if (!val.is_zero()) {
            return some_expr(mk_lit(literal(val - nat(1))));
        }
    }

    if (get_app_fn(t) == *g_nat_succ && get_app_num_args(t) == 1) {
        return some_expr(app_arg(t));
    }
    return none_expr();
}

lbool type_checker::is_def_eq_offset(expr const &t, expr const &s) {
    if (is_nat_zero(t) && is_nat_zero(s)) return l_true;
    optional<expr> pred_t = is_nat_succ(t);
    optional<expr> pred_s = is_nat_succ(s);
    if (pred_t && pred_s) {
        return to_lbool(is_def_eq_core(*pred_t, *pred_s));
```

```
    }
    return l_undef;
}

lbool type_checker::lazy_delta_reduction(expr &t_n, expr &s_n) {
    while (true) {
        lbool r = is_def_eq_offset(t_n, s_n);
        if (r != l_undef) return r;

        if (!has_fvar(t_n) && !has_fvar(s_n)) {
            if (auto t_v = reduce_nat(t_n)) {
                return to_lbool(is_def_eq_core(*t_v, s_n));
            } else if (auto s_v = reduce_nat(s_n)) {
                return to_lbool(is_def_eq_core(t_n, *s_v));
            }
        }

        if (auto t_v = reduce_native(env(), t_n)) {
            return to_lbool(is_def_eq_core(*t_v, s_n));
        } else if (auto s_v = reduce_native(env(), s_n)) {
            return to_lbool(is_def_eq_core(t_n, *s_v));
        }

        switch (lazy_delta_reduction_step(t_n, s_n)) {
            case reduction_status::Continue:
                break;
            case reduction_status::DefUnknown:
                return l_undef;
            case reduction_status::DefEqual:
                return l_true;
            case reduction_status::DefDiff:
                return l_false;
        }
    }
}

static expr *g_string_mk = nullptr;

lbool type_checker::try_string_lit_expansion_core(expr const &t,
                                                  expr const &s) {
    if (is_string_lit(t) && is_app(s) && app_fn(s) == *g_string_mk) {
        return to_lbool(is_def_eq_core(string_lit_to_constructor(t), s));
    }
    return l_undef;
}

lbool type_checker::try_string_lit_expansion(expr const &t, expr const &s) {
    lbool r = try_string_lit_expansion_core(t, s);
    if (r != l_undef) return r;
    return try_string_lit_expansion_core(s, t);
}

bool type_checker::is_def_eq_core(expr const &t, expr const &s) {
    check_system("is_definitionally_equal");
    bool use_hash = true;
    lbool r = quick_is_def_eq(t, s, use_hash);
    if (r != l_undef) return r == l_true;

    // apply whnf (without using delta-reduction or normalizer extensions)
    expr t_n = whnf_core(t);
    expr s_n = whnf_core(s);

    if (!is_eqp(t_n, t) || !is_eqp(s_n, s)) {
        r = quick_is_def_eq(t_n, s_n);
        if (r != l_undef) return r == l_true;
    }

    if (is_def_eq_proof_irrel(t_n, s_n)) return true;

    r = lazy_delta_reduction(t_n, s_n);
```

```
    if (r != l_undef) return r == l_true;

    if (is_constant(t_n) && is_constant(s_n) &&
        const_name(t_n) == const_name(s_n) &&
        is_def_eq(const_levels(t_n), const_levels(s_n)))
        return true;

    if (is_fvar(t_n) && is_fvar(s_n) && fvar_name(t_n) == fvar_name(s_n))
        return true;

    if (is_proj(t_n) && is_proj(s_n) && proj_idx(t_n) == proj_idx(s_n) &&
        is_def_eq(proj_expr(t_n), proj_expr(s_n)))
        return true;

    // At this point, t_n and s_n are in weak head normal form (modulo
    // metavariables and proof irrelevance)
    if (is_def_eq_app(t_n, s_n)) return true;

    if (try_eta_expansion(t_n, s_n)) return true;

    r = try_string_lit_expansion(t_n, s_n);
    if (r != l_undef) return r == l_true;

    return false;
}

bool type_checker::is_def_eq(expr const &t, expr const &s) {
    bool r = is_def_eq_core(t, s);
    if (r) m_st->m_eqv_manager.add_equiv(t, s);
    return r;
}

expr type_checker::eta_expand(expr const &e) {
    buffer<expr> fvars;
    flet<local_ctx> save_lctx(m_lctx, m_lctx);
    expr it = e;
    while (is_lambda(it)) {
        expr d =
            instantiate_rev(binding_domain(it), fvars.size(), fvars.data());
        fvars.push_back(m_lctx.mk_local_decl(m_st->m_ngen, binding_name(it), d,
                                             binding_info(it)));
        it = binding_body(it);
    }
    it = instantiate_rev(it, fvars.size(), fvars.data());
    expr it_type = whnf(infer(it));
    if (!is_pi(it_type)) return e;
    buffer<expr> args;
    while (is_pi(it_type)) {
        expr arg = m_lctx.mk_local_decl(m_st->m_ngen, binding_name(it_type),
                                        binding_domain(it_type),
                                        binding_info(it_type));
        args.push_back(arg);
        fvars.push_back(arg);
        it_type = whnf(instantiate(binding_body(it_type), arg));
    }
    expr r = mk_app(it, args);
    return m_lctx.mk_lambda(fvars, r);
}

type_checker::type_checker(environment const &env, local_ctx const &lctx,
                           bool safe_only)
    : m_st_owner(true),
      m_st(new state(env)),
      m_lctx(lctx),
      m_safe_only(safe_only),
      m_lparams(nullptr) {}

type_checker::type_checker(state &st, local_ctx const &lctx, bool safe_only)
    : m_st_owner(false),
      m_st(&st),
```

```
      m_lctx(lctx),
      m_safe_only(safe_only),
      m_lparams(nullptr) {}

type_checker::type_checker(type_checker &&src)
    : m_st_owner(src.m_st_owner),
      m_st(src.m_st),
      m_lctx(std::move(src.m_lctx)),
      m_safe_only(src.m_safe_only),
      m_lparams(src.m_lparams) {
    src.m_st_owner = false;
}

type_checker::~type_checker() {
    if (m_st_owner) delete m_st;
}

extern "C" uint8 lean_kernel_is_def_eq(lean_object *env, lean_object *lctx,
                                       lean_object *a, lean_object *b) {
    return type_checker(environment(env), local_ctx(lctx))
        .is_def_eq(expr(a), expr(b));
}

extern "C" lean_object *lean_kernel_whnf(lean_object *env, lean_object *lctx,
                                         lean_object *a) {
    return type_checker(environment(env), local_ctx(lctx))
        .whnf(expr(a))
        .steal();
}

void initialize_type_checker() {
    g_dont_care = new expr(mk_const("dontcare"));
    mark_persistent(g_dont_care->raw());
    g_kernel_fresh = new name("_kernel_fresh");
    mark_persistent(g_kernel_fresh->raw());
    g_nat_zero = new expr(mk_constant(name{"Nat", "zero"}));
    mark_persistent(g_nat_zero->raw());
    g_nat_succ = new expr(mk_constant(name{"Nat", "succ"}));
    mark_persistent(g_nat_succ->raw());
    g_nat_add = new expr(mk_constant(name{"Nat", "add"}));
    mark_persistent(g_nat_add->raw());
    g_nat_sub = new expr(mk_constant(name{"Nat", "sub"}));
    mark_persistent(g_nat_sub->raw());
    g_nat_mul = new expr(mk_constant(name{"Nat", "mul"}));
    mark_persistent(g_nat_mul->raw());
    g_nat_div = new expr(mk_constant(name{"Nat", "div"}));
    mark_persistent(g_nat_div->raw());
    g_nat_mod = new expr(mk_constant(name{"Nat", "mod"}));
    mark_persistent(g_nat_mod->raw());
    g_nat_beq = new expr(mk_constant(name{"Nat", "beq"}));
    mark_persistent(g_nat_beq->raw());
    g_nat_ble = new expr(mk_constant(name{"Nat", "ble"}));
    mark_persistent(g_nat_ble->raw());
    g_string_mk = new expr(mk_constant(name{"String", "mk"}));
    mark_persistent(g_string_mk->raw());
    g_lean_reduce_bool = new expr(mk_constant(name{"Lean", "reduceBool"}));
    mark_persistent(g_lean_reduce_bool->raw());
    g_lean_reduce_nat = new expr(mk_constant(name{"Lean", "reduceNat"}));
    mark_persistent(g_lean_reduce_nat->raw());
    register_name_generator_prefix(*g_kernel_fresh);
}

void finalize_type_checker() {
    delete g_dont_care;
    delete g_kernel_fresh;
    delete g_nat_succ;
    delete g_nat_zero;
    delete g_nat_add;
    delete g_nat_sub;
    delete g_nat_mul;
```

```
    delete g_nat_div;
    delete g_nat_mod;
    delete g_nat_beq;
    delete g_nat_ble;
    delete g_string_mk;
    delete g_lean_reduce_bool;
    delete g_lean_reduce_nat;
}
}  // namespace lean
```