

```

1 // ::::::::::::::
2 // abstract.h
3 // ::::::::::::::
4 /*
5 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
6 Released under Apache 2.0 license as described in the file LICENSE.
7
8 Author: Leonardo de Moura
9 */
10 #pragma once
11 #include <utility>
12
13 #include "kernel/expr.h"
14
15 namespace lean {
16 /** \brief Replace the free variables s[0], ..., s[n-1] in e with bound
17  * variables bvar(n-1), ..., bvar(0). */
18 expr abstract(expr const &e, unsigned n, expr const *s);
19 inline expr abstract(expr const &e, expr const &s) {
20     return abstract(e, 1, &s);
21 }
22 expr abstract(expr const &e, name const &n);
23
24 } // namespace lean
25 // ::::::::::::::
26 // cache_stack.h
27 // ::::::::::::::
28 /*
29 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
30 Released under Apache 2.0 license as described in the file LICENSE.
31
32 Author: Leonardo de Moura
33 */
34 #pragma once
35 #include <lean/debug.h>
36
37 #include <memory>
38 #include <vector>
39
40 /** \brief Macro for creating a stack of objects of type Cache in thread local
41  storage. The argument \c Arg is provided to every new instance of Cache. The
42  macro provides the helper class Cache_ref that "reuses" cache objects from
43  the stack.
44  */
45 #define MK_CACHE_STACK(Cache, Arg) \
46     struct Cache##_stack { \
47         unsigned m_top; \
48         std::vector<std::unique_ptr<Cache>> m_cache_stack; \
49         Cache##_stack() : m_top(0) {} \
50     }; \
51     MK_THREAD_LOCAL_GET_DEF(Cache##_stack, get_##Cache##_stack); \
52     class Cache##_ref { \
53         Cache *m_cache; \
54     public: \
55         Cache##_ref() { \
56             Cache##_stack &s = get_##Cache##_stack(); \
57             lean_assert(s.m_top <= s.m_cache_stack.size()); \
58             if (s.m_top == s.m_cache_stack.size()) \
59                 s.m_cache_stack.push_back( \
60                     std::unique_ptr<Cache>(new Cache(Arg))); \
61             m_cache = s.m_cache_stack[s.m_top].get(); \
62             s.m_top++; \
63         } \
64         ~Cache##_ref() { \
65             Cache##_stack &s = get_##Cache##_stack(); \
66             lean_assert(s.m_top > 0); \
67             s.m_top--; \
68             m_cache->clear(); \
69

```

```

70         }
71         Cache *operator->() const { return m_cache; }
72     };
73 // ::::::::::::::
74 // declaration.h
75 // ::::::::::::::
76 /*
77 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
78 Released under Apache 2.0 license as described in the file LICENSE.
79
80 Author: Leonardo de Moura
81 */
82 #pragma once
83 #include <algorithm>
84 #include <limits>
85 #include <string>
86
87 #include "kernel/expr.h"
88
89 namespace lean {
90 /**
91 inductive reducibility_hints
92 | opaque   : reducibility_hints
93 | abbrev   : reducibility_hints
94 | regular  : nat → reducibility_hints
95
96 Reducibility hints are used in the convertibility checker (aka is_def_eq
97 predicate), whenever checking a constraint such as
98
99         (f ...) == (g ...)
100
101 where f and g are definitions, and the checker has to decide which one will be
102 unfolded. If f (g) is Opaque, then g (f) is unfolded if it is also
103 not marked as Opaque. Else if f (g) is Abbreviation, then f (g) is unfolded if
104 g (f) is also not marked as Abbreviation. Else if f and g are Regular, then
105 we unfold the one with the biggest definitional height. Otherwise unfold both.
106
107 The definitional height is by default computed by the kernel. It only takes into
108 account other Regular definitions used in a definition.
109
110 Remark: the hint only affects performance. */
111 enum class reducibility_hints_kind { Opaque, Abbreviation, Regular };
112 class reducibility_hints : public object_ref {
113     reducibility_hints(b_obj_arg o, bool b) : object_ref(o, b) {}
114     explicit reducibility_hints(object *r) : object_ref(r) {}
115
116 public:
117     static reducibility_hints mk_opaque() {
118         return reducibility_hints(
119             box(static_cast<unsigned>(reducibility_hints_kind::Opaque)));
120     }
121     static reducibility_hints mk_abbreviation() {
122         return reducibility_hints(
123             box(static_cast<unsigned>(reducibility_hints_kind::Abbreviation)));
124     }
125     static reducibility_hints mk_regular(unsigned h);
126     reducibility_hints_kind kind() const {
127         return static_cast<reducibility_hints_kind>(obj_tag(raw()));
128     }
129     bool is_regular() const {
130         return kind() == reducibility_hints_kind::Regular;
131     }
132     unsigned get_height() const;
133 };
134
135 /** Given h1 and h2 the hints for definitions f1 and f2, then
136     result is
137     < 0 If f1 should be unfolded
138     == 0 If f1 and f2 should be unfolded
139     > 0 If f2 should be unfolded */

```

```

140 int compare(reducibility_hints const &h1, reducibility_hints const &h2);
141
142 /*
143 structure constant_val :=
144 (id : name) (lparams : list name) (type : expr)
145 */
146 class constant_val : public object_ref {
147     public:
148         constant_val(name const &n, names const &lparams, expr const &type);
149         constant_val(constant_val const &other) : object_ref(other) {}
150         constant_val(constant_val &&other) : object_ref(other) {}
151         constant_val &operator=(constant_val const &other) {
152             object_ref::operator=(other);
153             return *this;
154         }
155         constant_val &operator=(constant_val &&other) {
156             object_ref::operator=(other);
157             return *this;
158         }
159         name const &get_name() const {
160             return static_cast<name const &>(cnstr_get_ref(*this, 0));
161         }
162         names const &get_lparams() const {
163             return static_cast<names const &>(cnstr_get_ref(*this, 1));
164         }
165         expr const &get_type() const {
166             return static_cast<expr const &>(cnstr_get_ref(*this, 2));
167         }
168 };
169
170 /*
171 structure axiom_val extends constant_val :=
172 (is_unsafe : bool)
173 */
174 class axiom_val : public object_ref {
175     public:
176         axiom_val(name const &n, names const &lparams, expr const &type,
177             bool is_unsafe);
178         axiom_val(axiom_val const &other) : object_ref(other) {}
179         axiom_val(axiom_val &&other) : object_ref(other) {}
180         axiom_val &operator=(axiom_val const &other) {
181             object_ref::operator=(other);
182             return *this;
183         }
184         axiom_val &operator=(axiom_val &&other) {
185             object_ref::operator=(other);
186             return *this;
187         }
188         constant_val const &to_constant_val() const {
189             return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
190         }
191         name const &get_name() const { return to_constant_val().get_name(); }
192         names const &get_lparams() const { return to_constant_val().get_lparams(); }
193         expr const &get_type() const { return to_constant_val().get_type(); }
194         bool is_unsafe() const;
195 };
196
197 enum class definition_safety { unsafe, safe, partial };
198
199 /*
200 structure definition_val extends constant_val :=
201 (value : expr) (hints : reducibility_hints) (is_unsafe : bool)
202 */
203 class definition_val : public object_ref {
204     public:
205         definition_val(name const &n, names const &lparams, expr const &type,
206             expr const &val, reducibility_hints const &hints,
207             definition_safety safety);
208         definition_val(definition_val const &other) : object_ref(other) {}
209         definition_val(definition_val &&other) : object_ref(other) {}

```

```

210     definition_val &operator=(definition_val const &other) {
211         object_ref::operator=(other);
212         return *this;
213     }
214     definition_val &operator=(definition_val &&other) {
215         object_ref::operator=(other);
216         return *this;
217     }
218     constant_val const &to_constant_val() const {
219         return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
220     }
221     name const &get_name() const { return to_constant_val().get_name(); }
222     names const &get_lparams() const { return to_constant_val().get_lparams(); }
223     expr const &get_type() const { return to_constant_val().get_type(); }
224     expr const &get_value() const {
225         return static_cast<expr const &>(cnstr_get_ref(*this, 1));
226     }
227     reducibility_hints const &get_hints() const {
228         return static_cast<reducibility_hints const &>(cnstr_get_ref(*this, 2));
229     }
230     definition_safety get_safety() const;
231     bool is_unsafe() const { return get_safety() == definition_safety::unsafe; }
232 };
233 typedef list_ref<definition_val> definition_vals;
234
235 /*
236 structure theorem_val extends constant_val :=
237 (value : task expr)
238 */
239 class theorem_val : public object_ref {
240 public:
241     theorem_val(name const &n, names const &lparams, expr const &type,
242                 expr const &val);
243     theorem_val(theorem_val const &other) : object_ref(other) {}
244     theorem_val(theorem_val &&other) : object_ref(other) {}
245     theorem_val &operator=(theorem_val const &other) {
246         object_ref::operator=(other);
247         return *this;
248     }
249     theorem_val &operator=(theorem_val &&other) {
250         object_ref::operator=(other);
251         return *this;
252     }
253     constant_val const &to_constant_val() const {
254         return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
255     }
256     name const &get_name() const { return to_constant_val().get_name(); }
257     names const &get_lparams() const { return to_constant_val().get_lparams(); }
258     expr const &get_type() const { return to_constant_val().get_type(); }
259     expr const &get_value() const {
260         return static_cast<expr const &>(cnstr_get_ref(*this, 1));
261     }
262 };
263
264 /*
265 structure opaque_val extends constant_val :=
266 (value : expr)
267 */
268 class opaque_val : public object_ref {
269 public:
270     opaque_val(name const &n, names const &lparams, expr const &type,
271                expr const &val, bool is_unsafe);
272     opaque_val(opaque_val const &other) : object_ref(other) {}
273     opaque_val(opaque_val &&other) : object_ref(other) {}
274     opaque_val &operator=(opaque_val const &other) {
275         object_ref::operator=(other);
276         return *this;
277     }
278     opaque_val &operator=(opaque_val &&other) {
279         object_ref::operator=(other);

```

```

280         return *this;
281     }
282     constant_val const &to_constant_val() const {
283         return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
284     }
285     name const &get_name() const { return to_constant_val().get_name(); }
286     names const &get_lparams() const { return to_constant_val().get_lparams(); }
287     expr const &get_type() const { return to_constant_val().get_type(); }
288     expr const &get_value() const {
289         return static_cast<expr const &>(cnstr_get_ref(*this, 1));
290     }
291     bool is_unsafe() const;
292 };
293
294 /*
295 structure constructor :=
296 (id : name) (type : expr)
297 */
298 typedef pair_ref<name, expr> constructor;
299 inline name const &constructor_name(constructor const &c) { return c.fst(); }
300 inline expr const &constructor_type(constructor const &c) { return c.snd(); }
301 typedef list_ref<constructor> constructors;
302
303 /**
304 structure inductive_type where
305 (id : name) (type : expr) (cnstrs : list constructor)
306 */
307 class inductive_type : public object_ref {
308 public:
309     inductive_type(name const &id, expr const &type,
310                   constructors const &cnstrs);
311     inductive_type(inductive_type const &other) : object_ref(other) {}
312     inductive_type(inductive_type &&other) : object_ref(other) {}
313     inductive_type &operator=(inductive_type const &other) {
314         object_ref::operator=(other);
315         return *this;
316     }
317     inductive_type &operator=(inductive_type &&other) {
318         object_ref::operator=(other);
319         return *this;
320     }
321     name const &get_name() const {
322         return static_cast<name const &>(cnstr_get_ref(*this, 0));
323     }
324     expr const &get_type() const {
325         return static_cast<expr const &>(cnstr_get_ref(*this, 1));
326     }
327     constructors const &get_cnstrs() const {
328         return static_cast<constructors const &>(cnstr_get_ref(*this, 2));
329     }
330 };
331 typedef list_ref<inductive_type> inductive_types;
332
333 /*
334 inductive declaration
335 | axiom_decl      (val : axiom_val)
336 | defn_decl       (val : definition_val)
337 | thm_decl        (val : theorem_val)
338 | opaque_decl     (val : opaque_val)
339 | quot_decl       (id : name)
340 | mutual_defn_decl (defns : list definition_val) -- All definitions must be
341 marked as `unsafe` | induct_decl      (lparams : list name) (nparams : nat)
342 (types : list inductive_type) (is_unsafe : bool)
343 */
344 enum class declaration_kind {
345     Axiom,
346     Definition,
347     Theorem,
348     Opaque,
349     Quot,

```

```

350     MutualDefinition,
351     Inductive
352 };
353 class declaration : public object_ref {
354     object *get_val_obj() const { return cnstr_get(raw(), 0); }
355     object_ref const &to_val() const { return cnstr_get_ref(*this, 0); }
356
357 public:
358     declaration();
359     declaration(declaration const &other) : object_ref(other) {}
360     declaration(declaration &&other) : object_ref(other) {}
361     /* low-level constructors */
362     explicit declaration(object *o) : object_ref(o) {}
363     explicit declaration(b_obj_arg o, bool b) : object_ref(o, b) {}
364     explicit declaration(object_ref const &o) : object_ref(o) {}
365     declaration_kind kind() const {
366         return static_cast<declaration_kind>(obj_tag(raw()));
367     }
368
369     declaration &operator=(declaration const &other) {
370         object_ref::operator=(other);
371         return *this;
372     }
373     declaration &operator=(declaration &&other) {
374         object_ref::operator=(other);
375         return *this;
376     }
377
378     friend bool is_eqp(declaration const &d1, declaration const &d2) {
379         return d1.raw() == d2.raw();
380     }
381
382     bool is_definition() const {
383         return kind() == declaration_kind::Definition;
384     }
385     bool is_axiom() const { return kind() == declaration_kind::Axiom; }
386     bool is_theorem() const { return kind() == declaration_kind::Theorem; }
387     bool is_opaque() const { return kind() == declaration_kind::Opaque; }
388     bool is_mutual() const {
389         return kind() == declaration_kind::MutualDefinition;
390     }
391     bool is_inductive() const { return kind() == declaration_kind::Inductive; }
392     bool is_unsafe() const;
393     bool has_value() const { return is_theorem() || is_definition(); }
394
395     axiom_val const &to_axiom_val() const {
396         lean_assert(is_axiom());
397         return static_cast<axiom_val const &>(cnstr_get_ref(raw(), 0));
398     }
399     definition_val const &to_definition_val() const {
400         lean_assert(is_definition());
401         return static_cast<definition_val const &>(cnstr_get_ref(raw(), 0));
402     }
403     theorem_val const &to_theorem_val() const {
404         lean_assert(is_theorem());
405         return static_cast<theorem_val const &>(cnstr_get_ref(raw(), 0));
406     }
407     opaque_val const &to_opaque_val() const {
408         lean_assert(is_opaque());
409         return static_cast<opaque_val const &>(cnstr_get_ref(raw(), 0));
410     }
411     definition_vals const &to_definition_vals() const {
412         lean_assert(is_mutual());
413         return static_cast<definition_vals const &>(cnstr_get_ref(raw(), 0));
414     }
415 };
416
417 inline optional<declaration> none_declaration() {
418     return optional<declaration>();
419 }

```

```

420 inline optional<declaration> some_declaration(declaration const &o) {
421     return optional<declaration>(o);
422 }
423 inline optional<declaration> some_declaration(declaration &&o) {
424     return optional<declaration>(std::forward<declaration>(o));
425 }
426
427 declaration mk_definition(name const &n, names const &lparams, expr const &t,
428                          expr const &v, reducibility_hints const &hints,
429                          definition_safety safety = definition_safety::safe);
430 declaration mk_definition(environment const &env, name const &n,
431                          names const &lparams, expr const &t, expr const &v,
432                          definition_safety safety = definition_safety::safe);
433 declaration mk_opaque(name const &n, names const &lparams, expr const &t,
434                      expr const &v, bool unsafe);
435 declaration mk_axiom(name const &n, names const &lparams, expr const &t,
436                     bool unsafe = false);
437 declaration mk_inductive_decl(names const &lparams, nat const &nparams,
438                              inductive_types const &types, bool is_unsafe);
439
440 /** \brief Similar to mk_definition but infer the value of unsafe flag.
441     That is, set it to true if \c t or \c v contains a unsafe declaration. */
442 declaration mk_definition_inferring_unsafe(environment const &env,
443                                             name const &n, names const &lparams,
444                                             expr const &t, expr const &v,
445                                             reducibility_hints const &hints);
446 declaration mk_definition_inferring_unsafe(environment const &env,
447                                             name const &n, names const &lparams,
448                                             expr const &t, expr const &v);
449 /** \brief Similar to mk_axiom but infer the value of unsafe flag.
450     That is, set it to true if \c t or \c v contains a unsafe declaration. */
451 declaration mk_axiom_inferring_unsafe(environment const &env, name const &n,
452                                       names const &lparams, expr const &t);
453
454 /** \brief View for manipulating declaration.induct_decl constructor.
455     | induct_decl      (lparams : list name) (nparams : nat) (types : list
456     inductive_type) (is_unsafe : bool) */
457 class inductive_decl : public object_ref {
458 public:
459     inductive_decl(inductive_decl const &other) : object_ref(other) {}
460     inductive_decl(inductive_decl &&other) : object_ref(other) {}
461     inductive_decl(declaration const &d) : object_ref(d) {
462         lean_assert(d.is_inductive());
463     }
464     inductive_decl &operator=(inductive_decl const &other) {
465         object_ref::operator=(other);
466         return *this;
467     }
468     inductive_decl &operator=(inductive_decl &&other) {
469         object_ref::operator=(other);
470         return *this;
471     }
472     names const &get_lparams() const {
473         return static_cast<names const &>(cnstr_get_ref(raw(), 0));
474     }
475     nat const &get_nparams() const {
476         return static_cast<nat const &>(cnstr_get_ref(raw(), 1));
477     }
478     inductive_types const &get_types() const {
479         return static_cast<inductive_types const &>(cnstr_get_ref(raw(), 2));
480     }
481     bool is_unsafe() const;
482 };
483
484 /**
485 structure inductive_val extends constant_val where
486 (nparams : nat)      -- Number of parameters
487 (nindices : nat)     -- Number of indices
488 (all : list name)    -- List of all (including this one) inductive datatypes in
489 the mutual declaration containing this one (cnstrs : list name) -- List of all

```



```

490 constructors for this inductive datatype (is_rec : bool)          -- `tt` iff it is
491 recursive (is_unsafe : bool) (is_reflexive : bool)
492 */
493 class inductive_val : public object_ref {
494     public:
495         inductive_val(name const &n, names const &lparams, expr const &type,
496                     unsigned nparams, unsigned nindices, names const &all,
497                     names const &cnstrs, bool is_rec, bool is_unsafe,
498                     bool is_reflexive, bool is_nested);
499         inductive_val(inductive_val const &other) : object_ref(other) {}
500         inductive_val(inductive_val &&other) : object_ref(other) {}
501         inductive_val &operator=(inductive_val const &other) {
502             object_ref::operator=(other);
503             return *this;
504         }
505         inductive_val &operator=(inductive_val &&other) {
506             object_ref::operator=(other);
507             return *this;
508         }
509         constant_val const &to_constant_val() const {
510             return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
511         }
512         unsigned get_nparams() const {
513             return static_cast<nat const &>(cnstr_get_ref(*this, 1))
514                 .get_small_value();
515         }
516         unsigned get_nindices() const {
517             return static_cast<nat const &>(cnstr_get_ref(*this, 2))
518                 .get_small_value();
519         }
520         names const &get_all() const {
521             return static_cast<names const &>(cnstr_get_ref(*this, 3));
522         }
523         names const &get_cnstrs() const {
524             return static_cast<names const &>(cnstr_get_ref(*this, 4));
525         }
526         unsigned get_ncnstrs() const { return length(get_cnstrs()); }
527         bool is_rec() const;
528         bool is_unsafe() const;
529         bool is_reflexive() const;
530         bool is_nested() const;
531 };
532
533 /*
534 structure constructor_val extends constant_val :=
535 (induct : name) -- Inductive type this constructor is a member of
536 (cidx : nat) -- Constructor index (i.e., position in the inductive
537 declaration) (nparams : nat) -- Number of parameters in inductive datatype
538 `induct` (nfields : nat) -- Number of fields (i.e., arity - nparams)
539 (is_unsafe : bool)
540 */
541 class constructor_val : public object_ref {
542     public:
543         constructor_val(name const &n, names const &lparams, expr const &type,
544                     name const &induct, unsigned cidx, unsigned nparams,
545                     unsigned nfields, bool is_unsafe);
546         constructor_val(constructor_val const &other) : object_ref(other) {}
547         constructor_val(constructor_val &&other) : object_ref(other) {}
548         constructor_val &operator=(constructor_val const &other) {
549             object_ref::operator=(other);
550             return *this;
551         }
552         constructor_val &operator=(constructor_val &&other) {
553             object_ref::operator=(other);
554             return *this;
555         }
556         constant_val const &to_constant_val() const {
557             return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
558         }
559         name const &get_induct() const {

```



```

560         return static_cast<name const &>(cnstr_get_ref(*this, 1));
561     }
562     unsigned get_cidx() const {
563         return static_cast<nat const &>(cnstr_get_ref(*this, 2))
564             .get_small_value();
565     }
566     unsigned get_nparams() const {
567         return static_cast<nat const &>(cnstr_get_ref(*this, 3))
568             .get_small_value();
569     }
570     unsigned get_nfields() const {
571         return static_cast<nat const &>(cnstr_get_ref(*this, 4))
572             .get_small_value();
573     }
574     bool is_unsafe() const;
575 };
576
577 /*
578 structure recursor_rule :=
579 (cnstr : name) -- Reduction rule for this constructor
580 (nfields : nat) -- Number of fields (i.e., without counting inductive datatype
581 parameters) (rhs : expr) -- Right hand side of the reduction rule
582 */
583 class recursor_rule : public object_ref {
584 public:
585     recursor_rule(name const &cnstr, unsigned nfields, expr const &rhs);
586     recursor_rule(recursor_rule const &other) : object_ref(other) {}
587     recursor_rule(recursor_rule &&other) : object_ref(other) {}
588     recursor_rule &operator=(recursor_rule const &other) {
589         object_ref::operator=(other);
590         return *this;
591     }
592     recursor_rule &operator=(recursor_rule &&other) {
593         object_ref::operator=(other);
594         return *this;
595     }
596     name const &get_cnstr() const {
597         return static_cast<name const &>(cnstr_get_ref(*this, 0));
598     }
599     unsigned get_nfields() const {
600         return static_cast<nat const &>(cnstr_get_ref(*this, 1))
601             .get_small_value();
602     }
603     expr const &get_rhs() const {
604         return static_cast<expr const &>(cnstr_get_ref(*this, 2));
605     }
606 };
607
608 typedef list_ref<recursor_rule> recursor_rules;
609
610 /*
611 structure recursor_val extends constant_val :=
612 (all : list name) -- List of all inductive datatypes in the mutual
613 declaration that generated this recursor (nparams : nat) -- Number
614 of parameters (nindices : nat) -- Number of indices (nmotives : nat)
615 -- Number of motives (nminors : nat) -- Number of minor premises
616 (rules : list recursor_rule) -- A reduction for each constructor
617 (k : bool) -- It supports K-like reduction
618 (is_unsafe : bool)
619 */
620 class recursor_val : public object_ref {
621 public:
622     recursor_val(name const &n, names const &lparams, expr const &type,
623                 names const &all, unsigned nparams, unsigned nindices,
624                 unsigned nmotives, unsigned nminors,
625                 recursor_rules const &rules, bool k, bool is_unsafe);
626     recursor_val(recursor_val const &other) : object_ref(other) {}
627     recursor_val(recursor_val &&other) : object_ref(other) {}
628     recursor_val &operator=(recursor_val const &other) {
629         object_ref::operator=(other);

```

```

630         return *this;
631     }
632     recursor_val &operator=(recursor_val &&other) {
633         object_ref::operator=(other);
634         return *this;
635     }
636     constant_val const &to_constant_val() const {
637         return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
638     }
639     name const &get_name() const { return to_constant_val().get_name(); }
640     name const &get_induct() const { return get_name().get_prefix(); }
641     names const &get_all() const {
642         return static_cast<names const &>(cnstr_get_ref(*this, 1));
643     }
644     unsigned get_nparams() const {
645         return static_cast<nat const &>(cnstr_get_ref(*this, 2))
646             .get_small_value();
647     }
648     unsigned get_nindices() const {
649         return static_cast<nat const &>(cnstr_get_ref(*this, 3))
650             .get_small_value();
651     }
652     unsigned get_nmotives() const {
653         return static_cast<nat const &>(cnstr_get_ref(*this, 4))
654             .get_small_value();
655     }
656     unsigned get_nminors() const {
657         return static_cast<nat const &>(cnstr_get_ref(*this, 5))
658             .get_small_value();
659     }
660     unsigned get_major_idx() const {
661         return get_nparams() + get_nmotives() + get_nminors() + get_nindices();
662     }
663     recursor_rules const &get_rules() const {
664         return static_cast<recursor_rules const &>(cnstr_get_ref(*this, 6));
665     }
666     bool is_k() const;
667     bool is_unsafe() const;
668 };
669
670 enum class quot_kind { Type, Mk, Lift, Ind };
671
672 /*
673 inductive quot_kind
674 | type -- `quot`
675 | cnstr -- `quot.mk`
676 | lift -- `quot.lift`
677 | ind -- `quot.ind`
678
679 structure quot_val extends constant_val :=
680 (kind : quot_kind)
681 */
682 class quot_val : public object_ref {
683 public:
684     quot_val(name const &n, names const &lparams, expr const &type,
685             quot_kind k);
686     quot_val(quot_val const &other) : object_ref(other) {}
687     quot_val(quot_val &&other) : object_ref(other) {}
688     quot_val &operator=(quot_val const &other) {
689         object_ref::operator=(other);
690         return *this;
691     }
692     quot_val &operator=(quot_val &&other) {
693         object_ref::operator=(other);
694         return *this;
695     }
696     constant_val const &to_constant_val() const {
697         return static_cast<constant_val const &>(cnstr_get_ref(*this, 0));
698     }
699     name const &get_name() const { return to_constant_val().get_name(); }

```

```

700     names const &get_lparams() const { return to_constant_val().get_lparams(); }
701     expr const &get_type() const { return to_constant_val().get_type(); }
702     quot_kind get_quot_kind() const;
703 };
704
705 /*
706  -- Information associated with constant declarations. --
707 inductive constant_info
708 | axiom_info      (val : axiom_val)
709 | defn_info       (val : definition_val)
710 | thm_info        (val : theorem_val)
711 | opaque_info     (val : opaque_val)
712 | quot_info       (val : quot_val)
713 | induct_info     (val : inductive_val)
714 | cnstr_info      (val : constructor_val)
715 | rec_info        (val : recursor_val)
716 */
717 enum class constant_info_kind {
718     Axiom,
719     Definition,
720     Theorem,
721     Opaque,
722     Quot,
723     Inductive,
724     Constructor,
725     Recursor
726 };
727 class constant_info : public object_ref {
728     object *get_val_obj() const { return cnstr_get(raw(), 0); }
729     object_ref const &to_val() const { return cnstr_get_ref(*this, 0); }
730     constant_val const &to_constant_val() const {
731         return static_cast<constant_val const &>(cnstr_get_ref(to_val(), 0));
732     }
733
734 public:
735     constant_info();
736     constant_info(declaration const &d);
737     constant_info(definition_val const &v);
738     constant_info(quot_val const &v);
739     constant_info(inductive_val const &v);
740     constant_info(constructor_val const &v);
741     constant_info(recursor_val const &v);
742     constant_info(constant_info const &other) : object_ref(other) {}
743     constant_info(constant_info &&other) : object_ref(other) {}
744     explicit constant_info(b_obj_arg o, bool b) : object_ref(o, b) {}
745     explicit constant_info(obj_arg o) : object_ref(o) {}
746
747     constant_info_kind kind() const {
748         return static_cast<constant_info_kind>(cnstr_tag(raw()));
749     }
750
751     constant_info &operator=(constant_info const &other) {
752         object_ref::operator=(other);
753         return *this;
754     }
755     constant_info &operator=(constant_info &&other) {
756         object_ref::operator=(other);
757         return *this;
758     }
759
760     friend bool is_eqp(constant_info const &d1, constant_info const &d2) {
761         return d1.raw() == d2.raw();
762     }
763
764     bool is_unsafe() const;
765
766     bool is_definition() const {
767         return kind() == constant_info_kind::Definition;
768     }
769     bool is_axiom() const { return kind() == constant_info_kind::Axiom; }

```

```

770 bool is_theorem() const { return kind() == constant_info_kind::Theorem; }
771 bool is_opaque() const { return kind() == constant_info_kind::Opaque; }
772 bool is_inductive() const {
773     return kind() == constant_info_kind::Inductive;
774 }
775 bool is_constructor() const {
776     return kind() == constant_info_kind::Constructor;
777 }
778 bool is_recursor() const { return kind() == constant_info_kind::Recursor; }
779 bool is_quot() const { return kind() == constant_info_kind::Quot; }
780
781 name const &get_name() const { return to_constant_val().get_name(); }
782 names const &get_lparams() const { return to_constant_val().get_lparams(); }
783 unsigned get_num_lparams() const { return length(get_lparams()); }
784 expr const &get_type() const { return to_constant_val().get_type(); }
785 bool has_value(bool allow_opaque = false) const {
786     return is_theorem() || is_definition() || (allow_opaque && is_opaque());
787 }
788 reducibility_hints const &get_hints() const;
789
790 axiom_val const &to_axiom_val() const {
791     lean_assert(is_axiom());
792     return static_cast<axiom_val const &>(to_val());
793 }
794 definition_val const &to_definition_val() const {
795     lean_assert(is_definition());
796     return static_cast<definition_val const &>(to_val());
797 }
798 theorem_val const &to_theorem_val() const {
799     lean_assert(is_theorem());
800     return static_cast<theorem_val const &>(to_val());
801 }
802 opaque_val const &to_opaque_val() const {
803     lean_assert(is_opaque());
804     return static_cast<opaque_val const &>(to_val());
805 }
806 inductive_val const &to_inductive_val() const {
807     lean_assert(is_inductive());
808     return static_cast<inductive_val const &>(to_val());
809 }
810 constructor_val const &to_constructor_val() const {
811     lean_assert(is_constructor());
812     return static_cast<constructor_val const &>(to_val());
813 }
814 recursor_val const &to_recursor_val() const {
815     lean_assert(is_recursor());
816     return static_cast<recursor_val const &>(to_val());
817 }
818 quot_val const &to_quot_val() const {
819     lean_assert(is_quot());
820     return static_cast<quot_val const &>(to_val());
821 }
822
823 expr get_value(bool DEBUG_CODE(allow_opaque)) const {
824     lean_assert(has_value(allow_opaque));
825     if (is_theorem())
826         return to_theorem_val().get_value();
827     else
828         return static_cast<expr const &>(cnstr_get_ref(to_val(), 1));
829 }
830 expr get_value() const { return get_value(false); }
831 };
832
833 inline optional<constant_info> none_constant_info() {
834     return optional<constant_info>();
835 }
836 inline optional<constant_info> some_constant_info(constant_info const &o) {
837     return optional<constant_info>(o);
838 }
839 inline optional<constant_info> some_constant_info(constant_info &&o) {

```

```

840     return optional<constant_info>(std::forward<constant_info>(o));
841 }
842
843 static_assert(static_cast<unsigned>(declaration_kind::Axiom) ==
844               static_cast<unsigned>(constant_info_kind::Axiom),
845               "declaration vs constant_info tag mismatch");
846 static_assert(static_cast<unsigned>(declaration_kind::Definition) ==
847               static_cast<unsigned>(constant_info_kind::Definition),
848               "declaration vs constant_info tag mismatch");
849 static_assert(static_cast<unsigned>(declaration_kind::Theorem) ==
850               static_cast<unsigned>(constant_info_kind::Theorem),
851               "declaration vs constant_info tag mismatch");
852
853 void initialize_declaration();
854 void finalize_declaration();
855 } // namespace lean
856 // ::::::::::::::
857 // environment.h
858 // ::::::::::::::
859 /*
860 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
861 Released under Apache 2.0 license as described in the file LICENSE.
862
863 Author: Leonardo de Moura
864 */
865 #pragma once
866 #include <lean/optional.h>
867
868 #include <memory>
869 #include <utility>
870 #include <vector>
871
872 #include "kernel/declaration.h"
873 #include "kernel/expr.h"
874 #include "util/list.h"
875 #include "util/name_map.h"
876 #include "util/name_set.h"
877 #include "util/rb_map.h"
878 #include "util/rc.h"
879
880 #ifndef LEAN_BELIEVER_TRUST_LEVEL
881 /* If an environment E is created with a trust level >
882    LEAN_BELIEVER_TRUST_LEVEL, then we can add declarations to E without type
883    checking them. */
884 #define LEAN_BELIEVER_TRUST_LEVEL 1024
885 #endif
886
887 namespace lean {
888 class environment_extension {
889 public:
890     virtual ~environment_extension() {}
891 };
892
893 class environment : public object_ref {
894     friend class add_inductive_fn;
895
896     void check_name(name const &n) const;
897     void check_duplicated_univ_params(names ls) const;
898
899     void add_core(constant_info const &info);
900     void mark_quot_initialized();
901     environment add(constant_info const &info) const;
902     environment add_axiom(declaration const &d, bool check) const;
903     environment add_definition(declaration const &d, bool check) const;
904     environment add_theorem(declaration const &d, bool check) const;
905     environment add_opaque(declaration const &d, bool check) const;
906     environment add_mutual(declaration const &d, bool check) const;
907     environment add_quot() const;
908     environment add_inductive(declaration const &d) const;
909

```

```

910 public:
911     environment(unsigned trust_lvl = 0);
912     environment(environment const &other) : object_ref(other) {}
913     environment(environment &&other) : object_ref(other) {}
914     explicit environment(b_obj_arg o, bool b) : object_ref(o, b) {}
915     explicit environment(obj_arg o) : object_ref(o) {}
916     ~environment() {}
917
918     environment &operator=(environment const &other) {
919         object_ref::operator=(other);
920         return *this;
921     }
922     environment &operator=(environment &&other) {
923         object_ref::operator=(other);
924         return *this;
925     }
926
927     /** \brief Return the trust level of this environment. */
928     unsigned trust_lvl() const;
929
930     bool is_quot_initialized() const;
931
932     void set_main_module(name const &n);
933
934     name get_main_module() const;
935
936     /** \brief Return information for the constant with name \c n (if it is
937      * defined in this environment). */
938     optional<constant_info> find(name const &n) const;
939
940     /** \brief Return information for the constant with name \c n. Throws and
941      * exception if constant declaration does not exist in this environment. */
942     constant_info get(name const &n) const;
943
944     /** \brief Extends the current environment with the given declaration */
945     environment add(declaration const &d, bool check = true) const;
946
947     /** \brief Apply the function \c f to each constant */
948     void for_each_constant(
949         std::function<void(constant_info const &d)> const &f) const;
950
951     /** \brief Pointer equality */
952     friend bool is_eqp(environment const &e1, environment const &e2) {
953         return e1.raw() == e2.raw();
954     }
955
956     void display_stats() const;
957 };
958
959 void check_no_metavar_no_fvar(environment const &env, name const &n,
960                               expr const &e);
961
962 void initialize_environment();
963 void finalize_environment();
964 } // namespace lean
965 // ::::::::::::::
966 // equiv_manager.h
967 // ::::::::::::::
968 /*
969 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
970 Released under Apache 2.0 license as described in the file LICENSE.
971
972 Author: Leonardo de Moura
973 */
974 #pragma once
975 #include <vector>
976
977 #include "kernel/expr_maps.h"
978
979 namespace lean {

```

```

980 class equiv_manager {
981     typedef unsigned node_ref;
982
983     struct node {
984         node_ref m_parent;
985         unsigned m_rank;
986     };
987
988     std::vector<node> m_nodes;
989     expr_map<node_ref> m_to_node;
990     bool m_use_hash;
991
992     node_ref mk_node();
993     node_ref find(node_ref n);
994     void merge(node_ref n1, node_ref n2);
995     node_ref to_node(expr const &e);
996     bool is_equiv_core(expr const &e1, expr const &e2);
997
998     public:
999     equiv_manager() : m_use_hash(false) {}
1000     bool is_equiv(expr const &e1, expr const &e2, bool use_hash = false);
1001     void add_equiv(expr const &e1, expr const &e2);
1002 };
1003 } // namespace lean
1004 // ::::::::::::::
1005 // expr_cache.h
1006 // ::::::::::::::
1007 /*
1008 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
1009 Released under Apache 2.0 license as described in the file LICENSE.
1010
1011 Author: Leonardo de Moura
1012 */
1013 #pragma once
1014 #include <vector>
1015
1016 #include "kernel/expr.h"
1017
1018 namespace lean {
1019 /** \brief Cache for storing mappings from expressions to expressions.
1020
1021     \warning The insert(k, v) method overwrites any entry (k1, v1) when
1022     hash(k) == hash(k1)
1023 */
1024 class expr_cache {
1025     struct entry {
1026         optional<expr> m_expr;
1027         expr m_result;
1028     };
1029     unsigned m_capacity;
1030     std::vector<entry> m_cache;
1031     std::vector<unsigned> m_used;
1032
1033     public:
1034     expr_cache(unsigned c) : m_capacity(c), m_cache(c) {}
1035     void insert(expr const &e, expr const &v);
1036     expr *find(expr const &e);
1037     void clear();
1038 };
1039 } // namespace lean
1040 // ::::::::::::::
1041 // expr_eq_fn.h
1042 // ::::::::::::::
1043 /*
1044 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
1045 Released under Apache 2.0 license as described in the file LICENSE.
1046
1047 Author: Leonardo de Moura
1048 */
1049 #pragma once

```



```

1050
1051 namespace lean {
1052 class expr;
1053 // =====
1054 // Structural equality
1055 /** \brief Binder information is ignored in the following predicate */
1056 bool is_equal(expr const &a, expr const &b);
1057 inline bool operator==(expr const &a, expr const &b) { return is_equal(a, b); }
1058 inline bool operator!=(expr const &a, expr const &b) {
1059     return !operator==(a, b);
1060 }
1061 // =====
1062
1063 /** \brief Similar to ==, but it also compares binder information */
1064 bool is_bi_equal(expr const &a, expr const &b);
1065 struct is_bi_equal_proc {
1066     bool operator()(expr const &e1, expr const &e2) const {
1067         return is_bi_equal(e1, e2);
1068     }
1069 };
1070
1071 /** Similar to is_bi_equal_proc, but it has a flag that allows us to switch
1072  * select == or is_bi_equal */
1073 struct is_cond_bi_equal_proc {
1074     bool m_use_bi;
1075     is_cond_bi_equal_proc(bool b) : m_use_bi(b) {}
1076     bool operator()(expr const &e1, expr const &e2) const {
1077         return m_use_bi ? is_bi_equal(e1, e2) : e1 == e2;
1078     }
1079 };
1080 } // namespace lean
1081 // ::::::::::::::
1082 // expr.h
1083 // ::::::::::::::
1084 /*
1085 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
1086 Released under Apache 2.0 license as described in the file LICENSE.
1087
1088 Author: Leonardo de Moura
1089 */
1090 #pragma once
1091 #include <lean/hash.h>
1092 #include <lean/optional.h>
1093 #include <lean/serializer.h>
1094 #include <lean/thread.h>
1095
1096 #include <algorithm>
1097 #include <iostream>
1098 #include <limits>
1099 #include <string>
1100 #include <tuple>
1101 #include <utility>
1102
1103 #include "kernel/expr_eq_fn.h"
1104 #include "kernel/level.h"
1105 #include "util/buffer.h"
1106 #include "util/format.h"
1107 #include "util/kvmap.h"
1108 #include "util/list_fn.h"
1109 #include "util/name.h"
1110 #include "util/nat.h"
1111
1112 namespace lean {
1113 /** Binder annotations for Pi/lambda expressions */
1114 enum class binder_info { Default, Implicit, StrictImplicit, InstImplicit, Rec };
1115
1116 inline binder_info mk_binder_info() { return binder_info::Default; }
1117 inline binder_info mk_implicit_binder_info() { return binder_info::Implicit; }
1118 inline binder_info mk_strict_implicit_binder_info() {
1119     return binder_info::StrictImplicit;

```

```

1120 }
1121 inline binder_info mk_inst_implicit_binder_info() {
1122     return binder_info::InstImplicit;
1123 }
1124 inline binder_info mk_rec_info() { return binder_info::Rec; }
1125
1126 inline bool is_default(binder_info bi) { return bi == binder_info::Default; }
1127 inline bool is_implicit(binder_info bi) { return bi == binder_info::Implicit; }
1128 inline bool is_strict_implicit(binder_info bi) {
1129     return bi == binder_info::StrictImplicit;
1130 }
1131 inline bool is_inst_implicit(binder_info bi) {
1132     return bi == binder_info::InstImplicit;
1133 }
1134 inline bool is_explicit(binder_info bi) {
1135     return !is_implicit(bi) && !is_strict_implicit(bi) && !is_inst_implicit(bi);
1136 }
1137 inline bool is_rec(binder_info bi) { return bi == binder_info::Rec; }
1138
1139 /* Expression literal values */
1140 enum class literal_kind { Nat, String };
1141 class literal : public object_ref {
1142     explicit literal(b_obj_arg o, bool b) : object_ref(o, b) {}
1143
1144 public:
1145     explicit literal(char const *v);
1146     explicit literal(unsigned v);
1147     explicit literal(mpz const &v);
1148     explicit literal(nat const &v);
1149     literal() : literal(0u) {}
1150     literal(literal const &other) : object_ref(other) {}
1151     literal(literal &&other) : object_ref(other) {}
1152     literal &operator=(literal const &other) {
1153         object_ref::operator=(other);
1154         return *this;
1155     }
1156     literal &operator=(literal &&other) {
1157         object_ref::operator=(other);
1158         return *this;
1159     }
1160
1161     static literal_kind kind(object *o) {
1162         return static_cast<literal_kind>(cnstr_tag(o));
1163     }
1164     literal_kind kind() const { return kind(raw()); }
1165     string_ref const &get_string() const {
1166         lean_assert(kind() == literal_kind::String);
1167         return static_cast<string_ref const &>(cnstr_get_ref(*this, 0));
1168     }
1169     nat const &get_nat() const {
1170         lean_assert(kind() == literal_kind::Nat);
1171         return static_cast<nat const &>(cnstr_get_ref(*this, 0));
1172     }
1173     bool is_zero() const {
1174         return kind() == literal_kind::Nat && get_nat().is_zero();
1175     }
1176     friend bool operator==(literal const &a, literal const &b);
1177     friend bool operator<(literal const &a, literal const &b);
1178     void serialize(serializer &s) const { s.write_object(raw()); }
1179     static literal deserialize(deserializer &d) {
1180         return literal(d.read_object(), true);
1181     }
1182 };
1183 inline bool operator!=(literal const &a, literal const &b) { return !(a == b); }
1184 inline serializer &operator<<(serializer &s, literal const &l) {
1185     l.serialize(s);
1186     return s;
1187 }
1188 inline literal read_literal(deserializer &d) { return literal::deserialize(d); }
1189 inline deserializer &operator>>(deserializer &d, literal &l) {

```

```

1190     l = read_literal(d);
1191     return d;
1192 }
1193
1194 /* =====
1195     Expressions
1196
1197 inductive Expr
1198 | bvar      : Nat → Expr          -- bound variables
1199 | fvar      : Name → Expr         -- free variables
1200 | mvar      : Name → Expr         -- meta variables
1201 | sort      : Level → Expr        -- Sort
1202 | const     : Name → List Level → Expr -- constants
1203 | app       : Expr → Expr → Expr  -- application
1204 | lam       : Name → BinderInfo → Expr → Expr → Expr -- lambda abstraction
1205 | forallE   : Name → BinderInfo → Expr → Expr → Expr -- (dependent) arrow
1206 | letE      : Name → Expr → Expr → Expr → Expr -- let expressions
1207 | lit       : Literal → Expr      -- literals
1208 | mdata     : MData → Expr → Expr -- metadata
1209 | proj      : Name → Nat → Expr → Expr -- projection
1210 */
1211 enum class expr_kind {
1212     BVar,
1213     FVar,
1214     MVar,
1215     Sort,
1216     Const,
1217     App,
1218     Lambda,
1219     Pi,
1220     Let,
1221     Lit,
1222     MData,
1223     Proj
1224 };
1225 class expr : public object_ref {
1226     explicit expr(object_ref &o) : object_ref(o) {}
1227
1228     friend expr mk_lit(literal const &lit);
1229     friend expr mk_mdata(kvmap const &d, expr const &e);
1230     friend expr mk_proj(name const &s, nat const &idx, expr const &e);
1231     friend expr mk_bvar(nat const &idx);
1232     friend expr mk_mvar(name const &n);
1233     friend expr mk_fvar(name const &n);
1234     friend expr mk_const(name const &n, levels const &ls);
1235     friend expr mk_app(expr const &f, expr const &a);
1236     friend expr mk_sort(level const &l);
1237     friend expr mk_lambda(name const &n, expr const &t, expr const &e,
1238                           binder_info bi);
1239     friend expr mk_pi(name const &n, expr const &t, expr const &e,
1240                      binder_info bi);
1241     friend expr mk_let(name const &n, expr const &t, expr const &v,
1242                      expr const &b);
1243
1244 public:
1245     expr();
1246     expr(expr const &other) : object_ref(other) {}
1247     expr(expr &&other) : object_ref(other) {}
1248     explicit expr(b_obj_arg o, bool b) : object_ref(o, b) {}
1249     explicit expr(obj_arg o) : object_ref(o) {}
1250     static expr_kind kind(object *o) {
1251         return static_cast<expr_kind>(cnstr_tag(o));
1252     }
1253     expr_kind kind() const { return kind(raw()); }
1254
1255     expr &operator=(expr const &other) {
1256         object_ref::operator=(other);
1257         return *this;
1258     }
1259     expr &operator=(expr &&other) {

```

```

1260     object_ref::operator=(other);
1261     return *this;
1262 }
1263
1264 friend bool is_eqp(expr const &e1, expr const &e2) {
1265     return e1.raw() == e2.raw();
1266 }
1267 void serialize(serializer &s) const { s.write_object(raw()); }
1268 static expr deserialize(deserializer &d) {
1269     return expr(d.read_object(), true);
1270 }
1271 };
1272
1273 typedef list_ref<expr> exprs;
1274 typedef pair<expr, expr> expr_pair;
1275
1276 inline serializer &operator<<(serializer &s, expr const &e) {
1277     e.serialize(s);
1278     return s;
1279 }
1280 inline serializer &operator<<(serializer &s, exprs const &es) {
1281     es.serialize(s);
1282     return s;
1283 }
1284 inline expr read_expr(deserializer &d) { return expr::deserialize(d); }
1285 inline exprs read_exprs(deserializer &d) { return read_list_ref<expr>(d); }
1286 inline deserializer &operator>>(deserializer &d, expr &e) {
1287     e = read_expr(d);
1288     return d;
1289 }
1290
1291 inline optional<expr> none_expr() { return optional<expr>(); }
1292 inline optional<expr> some_expr(expr const &e) { return optional<expr>(e); }
1293 inline optional<expr> some_expr(expr &&e) {
1294     return optional<expr>(std::forward<expr>(e));
1295 }
1296
1297 inline bool is_eqp(optional<expr> const &a, optional<expr> const &b) {
1298     return static_cast<bool>(a) == static_cast<bool>(b) &&
1299         (!a || is_eqp(*a, *b));
1300 }
1301
1302 unsigned hash(expr const &e);
1303 bool has_expr_mvar(expr const &e);
1304 bool has_univ_mvar(expr const &e);
1305 inline bool has_mvar(expr const &e) {
1306     return has_expr_mvar(e) || has_univ_mvar(e);
1307 }
1308 bool has_fvar(expr const &e);
1309 bool has_univ_param(expr const &e);
1310 unsigned get_loose_bvar_range(expr const &e);
1311
1312 struct expr_hash {
1313     unsigned operator()(expr const &e) const { return hash(e); }
1314 };
1315 struct expr_pair_hash {
1316     unsigned operator()(expr_pair const &p) const {
1317         return hash(hash(p.first), hash(p.second));
1318     }
1319 };
1320 struct expr_pair_eq {
1321     bool operator()(expr_pair const &p1, expr_pair const &p2) const {
1322         return p1.first == p2.first && p1.second == p2.second;
1323     }
1324 };
1325
1326 // =====
1327 // Testers
1328 static expr_kind expr_kind_core(object *o) {
1329     return static_cast<expr_kind>(cnstr_tag(o));

```

```

1330 }
1331 inline bool is_bvar(expr const &e) { return e.kind() == expr_kind::BVar; }
1332 inline bool is_fvar_core(object *o) {
1333     return expr_kind_core(o) == expr_kind::FVar;
1334 }
1335 inline bool is_fvar(expr const &e) { return e.kind() == expr_kind::FVar; }
1336 inline bool is_const(expr const &e) { return e.kind() == expr_kind::Const; }
1337 inline bool is_mvar(expr const &e) { return e.kind() == expr_kind::MVar; }
1338 inline bool is_app(expr const &e) { return e.kind() == expr_kind::App; }
1339 inline bool is_lambda(expr const &e) { return e.kind() == expr_kind::Lambda; }
1340 inline bool is_pi(expr const &e) { return e.kind() == expr_kind::Pi; }
1341 inline bool is_let(expr const &e) { return e.kind() == expr_kind::Let; }
1342 inline bool is_sort(expr const &e) { return e.kind() == expr_kind::Sort; }
1343 inline bool is_lit(expr const &e) { return e.kind() == expr_kind::Lit; }
1344 inline bool is_mdata(expr const &e) { return e.kind() == expr_kind::MData; }
1345 inline bool is_proj(expr const &e) { return e.kind() == expr_kind::Proj; }
1346 inline bool is_binding(expr const &e) { return is_lambda(e) || is_pi(e); }
1347
1348 bool is_atomic(expr const &e);
1349 bool is_arrow(expr const &t);
1350 bool is_default_var_name(name const &n);
1351 // =====
1352
1353 // =====
1354 // Constructors
1355 expr mk_lit(literal const &lit);
1356 expr mk_mdata(kvmap const &d, expr const &e);
1357 expr mk_proj(name const &s, nat const &idx, expr const &e);
1358 inline expr mk_proj(name const &s, unsigned idx, expr const &e) {
1359     return mk_proj(s, nat(idx), e);
1360 }
1361 expr mk_bvar(nat const &idx);
1362 inline expr mk_bvar(unsigned idx) { return mk_bvar(nat(idx)); }
1363 expr mk_fvar(name const &n);
1364 expr mk_const(name const &n, levels const &ls);
1365 inline expr mk_const(name const &n) { return mk_const(n, levels()); }
1366 expr mk_mvar(name const &n);
1367 expr mk_app(expr const &f, expr const &a);
1368 expr mk_app(expr const &f, unsigned num_args, expr const *args);
1369 expr mk_app(unsigned num_args, expr const *args);
1370 inline expr mk_app(std::initializer_list<expr> const &l) {
1371     return mk_app(l.size(), l.begin());
1372 }
1373 inline expr mk_app(buffer<expr> const &args) {
1374     return mk_app(args.size(), args.data());
1375 }
1376 inline expr mk_app(expr const &f, buffer<expr> const &args) {
1377     return mk_app(f, args.size(), args.data());
1378 }
1379 expr mk_app(expr const &f, list<expr> const &args);
1380 inline expr mk_app(expr const &e1, expr const &e2, expr const &e3) {
1381     return mk_app({e1, e2, e3});
1382 }
1383 inline expr mk_app(expr const &e1, expr const &e2, expr const &e3,
1384     expr const &e4) {
1385     return mk_app({e1, e2, e3, e4});
1386 }
1387 inline expr mk_app(expr const &e1, expr const &e2, expr const &e3,
1388     expr const &e4, expr const &e5) {
1389     return mk_app({e1, e2, e3, e4, e5});
1390 }
1391 expr mk_rev_app(expr const &f, unsigned num_args, expr const *args);
1392 expr mk_rev_app(unsigned num_args, expr const *args);
1393 inline expr mk_rev_app(buffer<expr> const &args) {
1394     return mk_rev_app(args.size(), args.data());
1395 }
1396 inline expr mk_rev_app(expr const &f, buffer<expr> const &args) {
1397     return mk_rev_app(f, args.size(), args.data());
1398 }
1399 expr mk_lambda(name const &n, expr const &t, expr const &e,

```

```

1400         binder_info bi = mk_binder_info();
1401 expr mk_pi(name const &n, expr const &t, expr const &e,
1402         binder_info bi = mk_binder_info());
1403 inline expr mk_binding(expr_kind k, name const &n, expr const &t, expr const &e,
1404         binder_info bi = mk_binder_info()) {
1405     return k == expr_kind::Pi ? mk_pi(n, t, e, bi) : mk_lambda(n, t, e, bi);
1406 }
1407 expr mk_arrow(expr const &t, expr const &e);
1408 expr mk_let(name const &n, expr const &t, expr const &v, expr const &b);
1409 expr mk_sort(level const &l);
1410 expr mk_Prop();
1411 expr mk_Type();
1412 // =====
1413
1414 // =====
1415 // Accessors
1416 inline literal const &lit_value(expr const &e) {
1417     lean_assert(is_lit(e));
1418     return static_cast<literal const &>(cnstr_get_ref(e, 0));
1419 }
1420 inline bool is_nat_lit(expr const &e) {
1421     return is_lit(e) && lit_value(e).kind() == literal_kind::Nat;
1422 }
1423 inline bool is_string_lit(expr const &e) {
1424     return is_lit(e) && lit_value(e).kind() == literal_kind::String;
1425 }
1426 expr lit_type(literal const &e);
1427 inline kmap const &mdata_data(expr const &e) {
1428     lean_assert(is_mdata(e));
1429     return static_cast<kmap const &>(cnstr_get_ref(e, 0));
1430 }
1431 inline expr const &mdata_expr(expr const &e) {
1432     lean_assert(is_mdata(e));
1433     return static_cast<expr const &>(cnstr_get_ref(e, 1));
1434 }
1435 inline name const &proj_sname(expr const &e) {
1436     lean_assert(is_proj(e));
1437     return static_cast<name const &>(cnstr_get_ref(e, 0));
1438 }
1439 inline nat const &proj_idx(expr const &e) {
1440     lean_assert(is_proj(e));
1441     return static_cast<nat const &>(cnstr_get_ref(e, 1));
1442 }
1443 inline expr const &proj_expr(expr const &e) {
1444     lean_assert(is_proj(e));
1445     return static_cast<expr const &>(cnstr_get_ref(e, 2));
1446 }
1447 inline nat const &bvar_idx(expr const &e) {
1448     lean_assert(is_bvar(e));
1449     return static_cast<nat const &>(cnstr_get_ref(e, 0));
1450 }
1451 inline bool is_bvar(expr const &e, unsigned i) {
1452     return is_bvar(e) && bvar_idx(e) == i;
1453 }
1454 inline name const &fvar_name_core(object *o) {
1455     lean_assert(is_fvar_core(o));
1456     return static_cast<name const &>(cnstr_get_ref(o, 0));
1457 }
1458 inline name const &fvar_name(expr const &e) {
1459     lean_assert(is_fvar(e));
1460     return static_cast<name const &>(cnstr_get_ref(e, 0));
1461 }
1462 inline level const &sort_level(expr const &e) {
1463     lean_assert(is_sort(e));
1464     return static_cast<level const &>(cnstr_get_ref(e, 0));
1465 }
1466 inline name const &mvar_name(expr const &e) {
1467     lean_assert(is_mvar(e));
1468     return static_cast<name const &>(cnstr_get_ref(e, 0));
1469 }

```



```

1470 inline name const &const_name(expr const &e) {
1471     lean_assert(is_const(e));
1472     return static_cast<name const &>(cnstr_get_ref(e, 0));
1473 }
1474 inline levels const &const_levels(expr const &e) {
1475     lean_assert(is_const(e));
1476     return static_cast<levels const &>(cnstr_get_ref(e, 1));
1477 }
1478 inline bool is_const(expr const &e, name const &n) {
1479     return is_const(e) && const_name(e) == n;
1480 }
1481 inline expr const &app_fn(expr const &e) {
1482     lean_assert(is_app(e));
1483     return static_cast<expr const &>(cnstr_get_ref(e, 0));
1484 }
1485 inline expr const &app_arg(expr const &e) {
1486     lean_assert(is_app(e));
1487     return static_cast<expr const &>(cnstr_get_ref(e, 1));
1488 }
1489 inline name const &binding_name(expr const &e) {
1490     lean_assert(is_binding(e));
1491     return static_cast<name const &>(cnstr_get_ref(e, 0));
1492 }
1493 inline expr const &binding_domain(expr const &e) {
1494     lean_assert(is_binding(e));
1495     return static_cast<expr const &>(cnstr_get_ref(e, 1));
1496 }
1497 inline expr const &binding_body(expr const &e) {
1498     lean_assert(is_binding(e));
1499     return static_cast<expr const &>(cnstr_get_ref(e, 2));
1500 }
1501 binder_info binding_info(expr const &e);
1502 inline name const &let_name(expr const &e) {
1503     lean_assert(is_let(e));
1504     return static_cast<name const &>(cnstr_get_ref(e, 0));
1505 }
1506 inline expr const &let_type(expr const &e) {
1507     lean_assert(is_let(e));
1508     return static_cast<expr const &>(cnstr_get_ref(e, 1));
1509 }
1510 inline expr const &let_value(expr const &e) {
1511     lean_assert(is_let(e));
1512     return static_cast<expr const &>(cnstr_get_ref(e, 2));
1513 }
1514 inline expr const &let_body(expr const &e) {
1515     lean_assert(is_let(e));
1516     return static_cast<expr const &>(cnstr_get_ref(e, 3));
1517 }
1518 inline bool is_shared(expr const &e) { return !is_exclusive(e.raw()); }
1519 //
1520
1521 // =====
1522 // Update
1523 expr update_app(expr const &e, expr const &new_fn, expr const &new_arg);
1524 expr update_binding(expr const &e, expr const &new_domain,
1525     expr const &new_body);
1526 expr update_binding(expr const &e, expr const &new_domain, expr const &new_body,
1527     binder_info bi);
1528 expr update_sort(expr const &e, level const &new_level);
1529 expr update_const(expr const &e, levels const &new_levels);
1530 expr update_let(expr const &e, expr const &new_type, expr const &new_value,
1531     expr const &new_body);
1532 expr update_mdata(expr const &e, expr const &new_e);
1533 expr update_proj(expr const &e, expr const &new_e);
1534 // =====
1535
1536 /** \brief Given \c e of the form <tt>(...(f a1) ... an)</tt>, store a1 ... an
1537     in args. If \c e is not an application, then nothing is stored in args.
1538
1539     It returns the f. */

```



```

1540 expr const &get_app_args(expr const &e, buffer<expr> &args);
1541 /** \brief Similar to \c get_app_args, but stores at most num args.
1542     Examples:
1543     1) get_app_args_at_most(f a b c, 2, args);
1544     stores {b, c} in args and returns (f a)
1545
1546     2) get_app_args_at_most(f a b c, 4, args);
1547     stores {a, b, c} in args and returns f */
1548 expr const &get_app_args_at_most(expr const &e, unsigned num,
1549                                   buffer<expr> &args);
1550
1551 /** \brief Similar to \c get_app_args, but arguments are stored in reverse order
1552     in \c args. If e is of the form <tt>...(f a1) ... an</tt>, then the
1553     procedure stores [an, ..., a1] in \c args. */
1554 expr const &get_app_rev_args(expr const &e, buffer<expr> &args);
1555 /** \brief Given \c e of the form <tt>...(f a1) ... a_n</tt>, return \c f. If
1556     * \c e is not an application, then return \c e. */
1557 expr const &get_app_fn(expr const &e);
1558 /** \brief Given \c e of the form <tt>...(f a1) ... a_n</tt>, return \c n. If
1559     * \c e is not an application, then return 0. */
1560 unsigned get_app_num_args(expr const &e);
1561
1562 /** \brief Return true iff \c e is a metavariable or an application of a
1563     * metavariable */
1564 inline bool is_mvar_app(expr const &e) { return is_mvar(get_app_fn(e)); }
1565
1566 // =====
1567 // Loose bound variable management
1568
1569 /** \brief Return true iff the given expression has loose bound variables. */
1570 inline bool has_loose_bvars(expr const &e) {
1571     return get_loose_bvar_range(e) > 0;
1572 }
1573
1574 /** \brief Return true iff \c e contains the loose bound variable <tt>(var
1575     * i)</tt>. */
1576 bool has_loose_bvar(expr const &e, unsigned i);
1577
1578 /** \brief Lower the loose bound variables >= s in \c e by \c d. That is, a
1579     loose bound variable <tt>(var i)</tt> s.t. <tt>i >= s</tt> is mapped into
1580     <tt>(var i-d)</tt>.
1581
1582     \pre s >= d */
1583 expr lower_loose_bvars(expr const &e, unsigned s, unsigned d);
1584 expr lower_loose_bvars(expr const &e, unsigned d);
1585
1586 /** \brief Lift loose bound variables >= s in \c e by d. */
1587 expr lift_loose_bvars(expr const &e, unsigned s, unsigned d);
1588 expr lift_loose_bvars(expr const &e, unsigned d);
1589 // =====
1590
1591 // =====
1592 // Implicit argument inference
1593 /**
1594     \brief Given \c t of the form <tt>Pi (x_1 : A_1) ... (x_k : A_k), B</tt>,
1595     mark the first \c num_params as implicit if they are not already marked, and
1596     they occur in the remaining arguments. If \c strict is false, then we
1597     also mark it implicit if it occurs in \c B.
1598     */
1599 expr infer_implicit(expr const &t, unsigned num_params, bool strict);
1600 expr infer_implicit(expr const &t, bool strict);
1601 // =====
1602
1603 // =====
1604 // Low level (raw) printing
1605 std::ostream &operator<<(std::ostream &out, expr const &e);
1606 // =====
1607
1608 void initialize_expr();
1609 void finalize_expr();

```

```

1610
1611 /* ===== LEGACY ===== */
1612 inline bool has_expr_metavar(expr const &e) { return has_expr_mvar(e); }
1613 inline bool has_univ_metavar(expr const &e) { return has_univ_mvar(e); }
1614 inline bool has_metavar(expr const &e) { return has_mvar(e); }
1615 inline bool has_param_univ(expr const &e) { return has_univ_param(e); }
1616 inline bool is_var(expr const &e) { return is_bvar(e); }
1617 inline bool is_var(expr const &e, unsigned idx) { return is_bvar(e, idx); }
1618 inline bool is_metavar(expr const &e) { return is_mvar(e); }
1619 inline bool is_metavar_app(expr const &e) { return is_mvar_app(e); }
1620 inline expr mk_metavar(name const &n) { return mk_mvar(n); }
1621 inline expr mk_constant(name const &n, levels const &ls) {
1622     return mk_const(n, ls);
1623 }
1624 inline expr mk_constant(name const &n) { return mk_constant(n, levels()); }
1625 inline bool is_constant(expr const &e) { return is_const(e); }
1626 inline expr update_constant(expr const &e, levels const &new_levels) {
1627     return update_const(e, new_levels);
1628 }
1629 /** \brief Similar to \c has_expr_metavar, but ignores metavariables occurring
1630     in local constant types.
1631     It also returns the meta-variable application found in \c e. */
1632 optional<expr> has_expr_metavar_strict(expr const &e);
1633 inline bool is_constant(expr const &e, name const &n) { return is_const(e, n); }
1634 } // namespace lean
1635 // ::::::::::::::
1636 // expr_maps.h
1637 // ::::::::::::::
1638 /*
1639 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1640 Released under Apache 2.0 license as described in the file LICENSE.
1641
1642 Author: Leonardo de Moura
1643 */
1644 #pragma once
1645 #include <functional>
1646 #include <unordered_map>
1647
1648 #include "kernel/expr.h"
1649
1650 namespace lean {
1651 // Maps based on structural equality. That is, two keys are equal iff they are
1652 // structurally equal
1653 template <typename T>
1654 using expr_map =
1655     typename std::unordered_map<expr, T, expr_hash, std::equal_to<expr>>;
1656 // The following map also takes into account binder information
1657 template <typename T>
1658 using expr_bi_map =
1659     typename std::unordered_map<expr, T, expr_hash, is_bi_equal_proc>;
1660
1661 template <typename T>
1662 class expr_cond_bi_map
1663     : public std::unordered_map<expr, T, expr_hash, is_cond_bi_equal_proc> {
1664 public:
1665     expr_cond_bi_map(bool use_bi = false)
1666         : std::unordered_map<expr, T, expr_hash, is_cond_bi_equal_proc>(
1667             10, expr_hash(), is_cond_bi_equal_proc(use_bi)) {}
1668 };
1669 }; // namespace lean
1670 // ::::::::::::::
1671 // expr_sets.h
1672 // ::::::::::::::
1673 /*
1674 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1675 Released under Apache 2.0 license as described in the file LICENSE.
1676
1677 Author: Leonardo de Moura
1678 */
1679 #pragma once

```

```

1680 #include <lean/hash.h>
1681
1682 #include <functional>
1683 #include <unordered_set>
1684 #include <utility>
1685
1686 #include "kernel/expr.h"
1687
1688 namespace lean {
1689 typedef std::unordered_set<expr, expr_hash, std::equal_to<expr>> expr_set;
1690 }
1691 // ::::::::::::::
1692 // find_fn.h
1693 // ::::::::::::::
1694 /*
1695 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1696 Released under Apache 2.0 license as described in the file LICENSE.
1697
1698 Author: Leonardo de Moura
1699 */
1700 #pragma once
1701 #include "kernel/expr.h"
1702 #include "kernel/for_each_fn.h"
1703
1704 namespace lean {
1705 /** \brief Return a subexpression of \c e that satisfies the predicate \c p. */
1706 template <typename P>
1707 optional<expr> find(expr const &e, P p) {
1708     optional<expr> result;
1709     for_each(e, [&](expr const &e, unsigned offset) {
1710         if (result) {
1711             return false;
1712         } else if (p(e, offset)) {
1713             result = e;
1714             return false;
1715         } else {
1716             return true;
1717         }
1718     });
1719     return result;
1720 }
1721 } // namespace lean
1722 // ::::::::::::::
1723 // for_each_fn.h
1724 // ::::::::::::::
1725 /*
1726 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1727 Released under Apache 2.0 license as described in the file LICENSE.
1728
1729 Author: Leonardo de Moura
1730 */
1731 #pragma once
1732 #include <functional>
1733 #include <memory>
1734 #include <utility>
1735
1736 #include "kernel/expr.h"
1737 #include "kernel/expr_sets.h"
1738 #include "util/buffer.h"
1739
1740 namespace lean {
1741 /** \brief Expression visitor.
1742
1743     The argument \c f must be a lambda (function object) containing the method
1744
1745     <code>
1746     bool operator()(expr const & e, unsigned offset)
1747     </code>
1748
1749     The \c offset is the number of binders under which \c e occurs.

```

```

1750 */
1751 void for_each(expr const &e,
1752               std::function<bool(expr const &, unsigned)> &&f); // NOLINT
1753 } // namespace lean
1754 // ::::::::::::::
1755 // inductive.h
1756 // ::::::::::::::
1757 /*
1758 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
1759 Released under Apache 2.0 license as described in the file LICENSE.
1760
1761 Author: Leonardo de Moura
1762 */
1763 #pragma once
1764 #include "kernel/environment.h"
1765 #include "kernel/instantiate.h"
1766 namespace lean {
1767 /**\ brief Return recursor name for the given inductive datatype name */
1768 name mk_rec_name(name const &I);
1769
1770 /* Auxiliary function for to_cnstr_when_K */
1771 optional<expr> mk_nullary_cnstr(environment const &env, expr const &type,
1772                                unsigned num_params);
1773
1774 /* For datatypes that support K-axiom, given `e` an element of that type, we
1775    convert (if possible) to the default constructor. For example, if `e : a =
1776    a`, then this method returns `eq.refl a` */
1777 template <typename WHNF, typename INFER, typename IS_DEF_EQ>
1778 inline optional<expr> to_cnstr_when_K(environment const &env,
1779                                       recursor_val const &rval, expr const &e,
1780                                       WHNF const &whnf, INFER const &infer_type,
1781                                       IS_DEF_EQ const &is_def_eq) {
1782     lean_assert(rval.is_k());
1783     expr app_type = whnf(infer_type(e));
1784     expr const &app_type_I = get_app_fn(app_type);
1785     if (!is_constant(app_type_I) || const_name(app_type_I) != rval.get_induct())
1786         return none_expr(); // type incorrect
1787     if (has_expr_mvar(app_type)) {
1788         buffer<expr> app_type_args;
1789         get_app_args(app_type, app_type_args);
1790         for (unsigned i = rval.get_nparams(); i < app_type_args.size(); i++) {
1791             if (has_expr_metavar(app_type_args[i])) return none_expr();
1792         }
1793     }
1794     optional<expr> new_cnstr_app =
1795         mk_nullary_cnstr(env, app_type, rval.get_nparams());
1796     if (!new_cnstr_app) return none_expr();
1797     expr new_type = infer_type(*new_cnstr_app);
1798     if (!is_def_eq(app_type, new_type)) return none_expr();
1799     return some_expr(*new_cnstr_app);
1800 }
1801
1802 optional<recursor_rule> get_rec_rule_for(recursor_val const &rec_val,
1803                                         expr const &major);
1804
1805 expr nat_lit_to_constructor(expr const &e);
1806 expr string_lit_to_constructor(expr const &e);
1807
1808 template <typename WHNF, typename INFER, typename IS_DEF_EQ>
1809 inline optional<expr> inductive_reduce_rec(environment const &env,
1810                                           expr const &e, WHNF const &whnf,
1811                                           INFER const &infer_type,
1812                                           IS_DEF_EQ const &is_def_eq) {
1813     expr const &rec_fn = get_app_fn(e);
1814     if (!is_constant(rec_fn)) return none_expr();
1815     optional<constant_info> rec_info = env.find(const_name(rec_fn));
1816     if (!rec_info || !rec_info->is_recursor()) return none_expr();
1817     buffer<expr> rec_args;
1818     get_app_args(e, rec_args);
1819     recursor_val const &rec_val = rec_info->to_recursor_val();

```

```

1820     unsigned major_idx = rec_val.get_major_idx();
1821     if (major_idx >= rec_args.size())
1822         return none_expr(); // major premise is missing
1823     expr major = rec_args[major_idx];
1824     if (rec_val.is_k()) {
1825         if (optional<expr> c = to_cnstr_when_K(env, rec_val, major, whnf,
1826                                             infer_type, is_def_eq)) {
1827             major = *c;
1828         }
1829     }
1830     major = whnf(major);
1831     if (is_nat_lit(major)) major = nat_lit_to_constructor(major);
1832     if (is_string_lit(major)) major = string_lit_to_constructor(major);
1833     optional<recursor_rule> rule = get_rec_rule_for(rec_val, major);
1834     if (!rule) return none_expr();
1835     buffer<expr> major_args;
1836     get_app_args(major, major_args);
1837     if (rule->get_nfields() > major_args.size()) return none_expr();
1838     if (length(const_levels(rec_fn)) != length(rec_info->get_lparams()))
1839         return none_expr();
1840     expr rhs = instantiate_lparams(rule->get_rhs(), rec_info->get_lparams(),
1841                                   const_levels(rec_fn));
1842     /* apply parameters, motives and minor premises from recursor application.
1843     */
1844     rhs = mk_app(
1845         rhs,
1846         rec_val.get_nparams() + rec_val.get_nmotives() + rec_val.get_nminors(),
1847         rec_args.data());
1848     /* The number of parameters in the constructor is not necessarily
1849     equal to the number of parameters in the recursor when we have
1850     nested inductive types. */
1851     unsigned nparams = major_args.size() - rule->get_nfields();
1852     /* apply fields from major premise */
1853     rhs = mk_app(rhs, rule->get_nfields(), major_args.data() + nparams);
1854     if (rec_args.size() > major_idx + 1) {
1855         /* recursor application has more arguments after major premise */
1856         unsigned nextra = rec_args.size() - major_idx - 1;
1857         rhs = mk_app(rhs, nextra, rec_args.data() + major_idx + 1);
1858     }
1859     return some_expr(rhs);
1860 }
1861
1862 template <typename WHNF, typename IS_STUCK>
1863 optional<expr> inductive_is_stuck(environment const &env, expr const &e,
1864                                   WHNF const &whnf, IS_STUCK const &is_stuck) {
1865     expr const &rec_fn = get_app_fn(e);
1866     if (!is_constant(rec_fn)) return none_expr();
1867     optional<constant_info> rec_info = env.find(const_name(rec_fn));
1868     if (!rec_info || !rec_info->is_recursor()) return none_expr();
1869     buffer<expr> rec_args;
1870     get_app_args(e, rec_args);
1871     recursor_val const &rec_val = rec_info->to_recursor_val();
1872     unsigned major_idx = rec_val.get_major_idx();
1873     if (rec_args.size() < major_idx + 1) return none_expr();
1874     expr cnstr_app = whnf(rec_args[major_idx]);
1875     if (rec_val.is_k()) {
1876         /* TODO(Leo): make it more precise. Remark: this piece of
1877         code does not affect the correctness of the kernel, but the
1878         effectiveness of the elaborator. */
1879         return none_expr();
1880     } else {
1881         return is_stuck(cnstr_app);
1882     }
1883 }
1884
1885 void initialize_inductive();
1886 void finalize_inductive();
1887 } // namespace lean
1888 // ::::::::::::::
1889 // init_module.h

```

```

1890 // ::::::::::::::
1891 /*
1892 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
1893 Released under Apache 2.0 license as described in the file LICENSE.
1894
1895 Author: Leonardo de Moura
1896 */
1897 #pragma once
1898 namespace lean {
1899 void initialize_kernel_module();
1900 void finalize_kernel_module();
1901 } // namespace lean
1902 // ::::::::::::::
1903 // instantiate.h
1904 // ::::::::::::::
1905 /*
1906 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1907 Released under Apache 2.0 license as described in the file LICENSE.
1908
1909 Author: Leonardo de Moura
1910 */
1911 #pragma once
1912 #include <functional>
1913
1914 #include "kernel/expr.h"
1915
1916 namespace lean {
1917 class ro_metavar_env;
1918 /** \brief Replace the loose bound variables with indices 0, ..., n-1 with s[0],
1919  * ..., s[n-1] in e. */
1920 expr instantiate(expr const &e, unsigned n, expr const *s);
1921 expr instantiate(expr const &e, std::initializer_list<expr> const &l);
1922 /** \brief Replace loose bound variable \c i with \c s in \c e. */
1923 expr instantiate(expr const &e, unsigned i, expr const &s);
1924 /** \brief Replace loose bound variable \c 0 with \c s in \c e. */
1925 expr instantiate(expr const &e, expr const &s);
1926
1927 /** \brief Replace the free variables with indices 0, ..., n-1 with s[n-1], ...,
1928  * s[0] in e. */
1929 expr instantiate_rev(expr const &e, unsigned n, expr const *s);
1930 inline expr instantiate_rev(expr const &e, buffer<expr> const &s) {
1931     return instantiate_rev(e, s.size(), s.data());
1932 }
1933
1934 expr apply_beta(expr f, unsigned num_rev_args, expr const *rev_args);
1935 bool is_head_beta(expr const &t);
1936 expr head_beta_reduce(expr const &t);
1937 /* If `e` is of the form `(fun x, t) a` return `head_beta_const_fn(t)` if `t`
1938  does not depend on `x`,
1939  and `e` otherwise. We also reduce `(fun x_1 ... x_n, x_i) a_1 ... a_n` into
1940  `a_{n-i-1}` */
1941 expr cheap_beta_reduce(expr const &e);
1942
1943 /** \brief Instantiate the universe level parameters \c ps occurring in \c e
1944  with the levels \c ls. \pre length(ps) == length(ls) */
1945 expr instantiate_lparams(expr const &e, names const &ps, levels const &ls);
1946
1947 class constant_info;
1948 /** \brief Instantiate the universe level parameters of the type of the given
1949  constant. \pre d.get_num_lparams() == length(ls) */
1950 expr instantiate_type_lparams(constant_info const &info, levels const &ls);
1951 /** \brief Instantiate the universe level parameters of the value of the given
1952  constant. \pre d.get_num_lparams() == length(ls) */
1953 expr instantiate_value_lparams(constant_info const &info, levels const &ls);
1954 } // namespace lean
1955 // ::::::::::::::
1956 // kernel_exception.h
1957 // ::::::::::::::
1958 /*
1959 Copyright (c) 2013 Microsoft Corporation. All rights reserved.

```



```

1960 Released under Apache 2.0 license as described in the file LICENSE.
1961
1962 Author: Leonardo de Moura
1963 */
1964 #pragma once
1965 #include "kernel/environment.h"
1966 #include "kernel/local_ctx.h"
1967
1968 namespace lean {
1969 /** \brief Base class for all kernel exceptions. */
1970 class kernel_exception : public exception {
1971     protected:
1972         environment m_env;
1973
1974     public:
1975         kernel_exception(environment const &env)
1976             : exception("kernel exception"), m_env(env) {}
1977         kernel_exception(environment const &env, char const *msg)
1978             : exception(msg), m_env(env) {}
1979         kernel_exception(environment const &env, sstream const &strm)
1980             : exception(strm), m_env(env) {}
1981         environment const &get_environment() const { return m_env; }
1982         environment const &env() const { return m_env; }
1983 };
1984
1985 class unknown_constant_exception : public kernel_exception {
1986     name m_name;
1987
1988     public:
1989         unknown_constant_exception(environment const &env, name const &n)
1990             : kernel_exception(env), m_name(n) {}
1991         name const &get_name() const { return m_name; }
1992 };
1993
1994 class already_declared_exception : public kernel_exception {
1995     name m_name;
1996
1997     public:
1998         already_declared_exception(environment const &env, name const &n)
1999             : kernel_exception(env), m_name(n) {}
2000         name const &get_name() const { return m_name; }
2001 };
2002
2003 class definition_type_mismatch_exception : public kernel_exception {
2004     declaration m_decl;
2005     expr m_given_type;
2006
2007     public:
2008         definition_type_mismatch_exception(environment const &env,
2009                                           declaration const &decl,
2010                                           expr const &given_type)
2011             : kernel_exception(env), m_decl(decl), m_given_type(given_type) {}
2012         declaration const &get_declaration() const { return m_decl; }
2013         expr const &get_given_type() const { return m_given_type; }
2014 };
2015
2016 class declaration_has_metavars_exception : public kernel_exception {
2017     name m_name;
2018     expr m_expr;
2019
2020     public:
2021         declaration_has_metavars_exception(environment const &env, name const &n,
2022                                           expr const &e)
2023             : kernel_exception(env), m_name(n), m_expr(e) {}
2024         name const &get_decl_name() const { return m_name; }
2025         expr const &get_expr() const { return m_expr; }
2026 };
2027
2028 class declaration_has_free_vars_exception : public kernel_exception {
2029     name m_name;

```



```

2030     expr m_expr;
2031
2032 public:
2033     declaration_has_free_vars_exception(environment const &env, name const &n,
2034                                         expr const &e)
2035         : kernel_exception(env), m_name(n), m_expr(e) {}
2036     name const &get_decl_name() const { return m_name; }
2037     expr const &get_expr() const { return m_expr; }
2038 };
2039
2040 class kernel_exception_with_lctx : public kernel_exception {
2041     local_ctx m_lctx;
2042
2043 public:
2044     kernel_exception_with_lctx(environment const &env, local_ctx const &lctx)
2045         : kernel_exception(env), m_lctx(lctx) {}
2046     local_ctx const &get_local_ctx() const { return m_lctx; }
2047 };
2048
2049 class function_expected_exception : public kernel_exception_with_lctx {
2050     expr m_fn;
2051
2052 public:
2053     function_expected_exception(environment const &env, local_ctx const &lctx,
2054                                expr const &fn)
2055         : kernel_exception_with_lctx(env, lctx), m_fn(fn) {}
2056     expr const &get_fn() const { return m_fn; }
2057 };
2058
2059 class type_expected_exception : public kernel_exception_with_lctx {
2060     expr m_type;
2061
2062 public:
2063     type_expected_exception(environment const &env, local_ctx const &lctx,
2064                             expr const &type)
2065         : kernel_exception_with_lctx(env, lctx), m_type(type) {}
2066     expr const &get_type() const { return m_type; }
2067 };
2068
2069 class type_mismatch_exception : public kernel_exception_with_lctx {
2070     expr m_given_type;
2071     expr m_expected_type;
2072
2073 public:
2074     type_mismatch_exception(environment const &env, local_ctx const &lctx,
2075                             expr const &given_type, expr const &expected_type)
2076         : kernel_exception_with_lctx(env, lctx),
2077           m_given_type(given_type),
2078           m_expected_type(expected_type) {}
2079     expr const &get_given_type() const { return m_given_type; }
2080     expr const &get_expected_type() const { return m_expected_type; }
2081 };
2082
2083 class def_type_mismatch_exception : public type_mismatch_exception {
2084     name m_name;
2085
2086 public:
2087     def_type_mismatch_exception(environment const &env, local_ctx const &lctx,
2088                                name const &n, expr const &given_type,
2089                                expr const &expected_type)
2090         : type_mismatch_exception(env, lctx, given_type, expected_type),
2091           m_name(n) {}
2092     name const &get_name() const { return m_name; }
2093 };
2094
2095 class expr_type_mismatch_exception : public kernel_exception_with_lctx {
2096     expr m_expr;
2097     expr m_expected_type;
2098
2099 public:

```

```

2100     expr_type_mismatch_exception(environment const &env, local_ctx const &lctx,
2101                                   expr const &e, expr const &expected_type)
2102     : kernel_exception_with_lctx(env, lctx),
2103       m_expr(e),
2104       m_expected_type(expected_type) {}
2105     expr const &get_expr() const { return m_expr; }
2106     expr const &get_expected_type() const { return m_expected_type; }
2107 };
2108
2109 class app_type_mismatch_exception : public kernel_exception_with_lctx {
2110     expr m_app;
2111     expr m_function_type;
2112     expr m_arg_type;
2113
2114 public:
2115     app_type_mismatch_exception(environment const &env, local_ctx const &lctx,
2116                                   expr const &app, expr const &function_type,
2117                                   expr const &arg_type)
2118     : kernel_exception_with_lctx(env, lctx),
2119       m_app(app),
2120       m_function_type(function_type),
2121       m_arg_type(arg_type) {}
2122     expr const &get_app() const { return m_app; }
2123     expr const &get_function_type() const { return m_function_type; }
2124     expr const &get_arg_type() const { return m_arg_type; }
2125 };
2126
2127 class invalid_proj_exception : public kernel_exception_with_lctx {
2128     expr m_proj;
2129
2130 public:
2131     invalid_proj_exception(environment const &env, local_ctx const &lctx,
2132                                   expr const &proj)
2133     : kernel_exception_with_lctx(env, lctx), m_proj(proj) {}
2134     expr const &get_proj() const { return m_proj; }
2135 };
2136
2137 /*
2138 Helper function for interfacing C++ code with code written in Lean.
2139 It executes closure `f` which produces an object_ref of type `A` and may throw
2140 an `kernel_exception` or `exception`. Then, convert result into `Except
2141 KernelException T` where `T` is the type of the lean objected represented by
2142 `A`. We use the constructor `KernelException.other <msg>` to handle C++
2143 `exception` objects which are not `kernel_exception`.
2144 ```
2145 inductive KernelException
2146 0 | unknownConstant (env : Environment) (name : Name)
2147 1 | alreadyDeclared (env : Environment) (name : Name)
2148 2 | declTypeMismatch (env : Environment) (decl : Declaration) (givenType :
2149 Expr) 3 | declHasMVars (env : Environment) (name : Name) (expr : Expr) 4 |
2150 declHasFVars (env : Environment) (name : Name) (expr : Expr) 5 |
2151 funExpected (env : Environment) (lctx : LocalContext) (expr : Expr) 6 |
2152 typeExpected (env : Environment) (lctx : LocalContext) (expr : Expr) 7 |
2153 letTypeMismatch (env : Environment) (lctx : LocalContext) (name : Name)
2154 (givenType : Expr) (expectedType : Expr) 8 | exprTypeMismatch (env :
2155 Environment) (lctx : LocalContext) (expr : Expr) (expectedType : Expr) 9 |
2156 appTypeMismatch (env : Environment) (lctx : LocalContext) (app : Expr) (funType
2157 : Expr) (argType : Expr) 10 | invalidProj (env : Environment) (lctx :
2158 LocalContext) (proj : Expr) 11 | other (msg : String)
2159 ```
2160 */
2161 template <typename A>
2162 object *catch_kernel_exceptions(std::function<A()> const &f) {
2163     try {
2164         A a = f();
2165         return mk_cnstr(1, a).steal();
2166     } catch (unknown_constant_exception &ex) {
2167         // 0 | unknownConstant (env : Environment) (name : Name)
2168         return mk_cnstr(0, mk_cnstr(0, ex.env(), ex.get_name())).steal();
2169     } catch (already_declared_exception &ex) {

```

```

2170     // 1 | alreadyDeclared (env : Environment) (name : Name)
2171     return mk_cnstr(0, mk_cnstr(1, ex.env(), ex.get_name())).steal();
2172 } catch (definition_type_mismatch_exception &ex) {
2173     // 2 | declTypeMismatch (env : Environment) (decl : Declaration)
2174     // (givenType : Expr)
2175     return mk_cnstr(0, mk_cnstr(2, ex.env(), ex.get_declaration(),
2176                                ex.get_given_type()))
2177         .steal();
2178 } catch (declaration_has_metavars_exception &ex) {
2179     // 3 | declHasMVars      (env : Environment) (name : Name) (expr : Expr)
2180     return mk_cnstr(
2181         0, mk_cnstr(3, ex.env(), ex.get_decl_name(), ex.get_expr()))
2182         .steal();
2183 } catch (declaration_has_free_vars_exception &ex) {
2184     // 4 | declHasFVars      (env : Environment) (name : Name) (expr : Expr)
2185     return mk_cnstr(
2186         0, mk_cnstr(4, ex.env(), ex.get_decl_name(), ex.get_expr()))
2187         .steal();
2188 } catch (function_expected_exception &ex) {
2189     // 5 | funExpected       (env : Environment) (lctx : LocalContext) (expr
2190     // : Expr)
2191     return mk_cnstr(0,
2192                    mk_cnstr(5, ex.env(), ex.get_local_ctx(), ex.get_fn()))
2193         .steal();
2194 } catch (type_expected_exception &ex) {
2195     // 6 | typeExpected      (env : Environment) (lctx : LocalContext) (expr
2196     // : Expr)
2197     return mk_cnstr(
2198         0, mk_cnstr(6, ex.env(), ex.get_local_ctx(), ex.get_type()))
2199         .steal();
2200 } catch (def_type_mismatch_exception &ex) {
2201     // 7 | letTypeMismatch   (env : Environment) (lctx : LocalContext) (name
2202     // : Name) (givenType : Expr) (expectedType : Expr)
2203     return mk_cnstr(0,
2204                    mk_cnstr(7, ex.env(), ex.get_local_ctx(), ex.get_name(),
2205                               ex.get_given_type(), ex.get_expected_type()))
2206         .steal();
2207 } catch (expr_type_mismatch_exception &ex) {
2208     // 8 | exprTypeMismatch (env : Environment) (lctx : LocalContext) (expr
2209     // : Expr) (expectedType : Expr)
2210     return mk_cnstr(0, mk_cnstr(8, ex.env(), ex.get_local_ctx(),
2211                                ex.get_expr(), ex.get_expected_type()))
2212         .steal();
2213 } catch (app_type_mismatch_exception &ex) {
2214     // 9 | appTypeMismatch   (env : Environment) (lctx : LocalContext) (app
2215     // : Expr) (funType : Expr) (argType : Expr)
2216     return mk_cnstr(0,
2217                    mk_cnstr(9, ex.env(), ex.get_local_ctx(), ex.get_app(),
2218                               ex.get_function_type(), ex.get_arg_type()))
2219         .steal();
2220 } catch (invalid_proj_exception &ex) {
2221     // 10 | invalidProj      (env : Environment) (lctx : LocalContext) (proj
2222     // : Expr)
2223     return mk_cnstr(
2224         0, mk_cnstr(10, ex.env(), ex.get_local_ctx(), ex.get_proj()))
2225         .steal();
2226 } catch (exception &ex) {
2227     // 11 | other           (msg : String)
2228     return mk_cnstr(0, mk_cnstr(11, string_ref(ex.what()))).steal();
2229 }
2230 }
2231 } // namespace lean
2232 // ::::::::::::::
2233 // level.h
2234 // ::::::::::::::
2235 /*
2236 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
2237 Released under Apache 2.0 license as described in the file LICENSE.
2238
2239 Author: Leonardo de Moura

```

```

2240 */
2241 #pragma once
2242 #include <lean/optional.h>
2243
2244 #include <algorithm>
2245 #include <iostream>
2246 #include <utility>
2247
2248 #include "util/format.h"
2249 #include "util/list_ref.h"
2250 #include "util/name.h"
2251 #include "util/options.h"
2252
2253 namespace lean {
2254 class environment;
2255 struct level_cell;
2256 /**
2257 inductive level
2258 | zero   : level
2259 | succ   : level → level
2260 | max    : level → level → level
2261 | imax   : level → level → level
2262 | param  : name → level
2263 | mvar   : name → level
2264
2265 We level.imax to handle Pi-types.
2266 */
2267 enum class level_kind { Zero, Succ, Max, IMax, Param, MVar };
2268
2269 /** \brief Universe level. */
2270 class level : public object_ref {
2271     friend level mk_succ(level const &l);
2272     friend level mk_max_core(level const &l1, level const &l2);
2273     friend level mk_imax_core(level const &l1, level const &l2);
2274     friend level mk_univ_param(name const &n);
2275     friend level mk_univ_mvar(name const &n);
2276     explicit level(object_ref &o) : object_ref(o) {}
2277
2278 public:
2279     /** \brief Universe zero */
2280     level();
2281     explicit level(obj_arg o) : object_ref(o) {}
2282     explicit level(b_obj_arg o, bool b) : object_ref(o, b) {}
2283     level(level const &other) : object_ref(other) {}
2284     level(level &&other) : object_ref(other) {}
2285     level_kind kind() const {
2286         return static_cast<level_kind>(lean_ptr_tag(raw()));
2287     }
2288     unsigned hash() const;
2289
2290     level &operator=(level const &other) {
2291         object_ref::operator=(other);
2292         return *this;
2293     }
2294     level &operator=(level &&other) {
2295         object_ref::operator=(other);
2296         return *this;
2297     }
2298
2299     friend bool is_eqp(level const &l1, level const &l2) {
2300         return l1.raw() == l2.raw();
2301     }
2302     void serialize(serializer &s) const { s.write_object(raw()); }
2303     static level deserialize(deserializer &d) {
2304         return level(d.read_object(), true);
2305     }
2306
2307     bool is_zero() const { return kind() == level_kind::Zero; }
2308     bool is_succ() const { return kind() == level_kind::Succ; }
2309     bool is_max() const { return kind() == level_kind::Max; }

```

```

2310     bool is_imax() const { return kind() == level_kind::IMax; }
2311     bool is_param() const { return kind() == level_kind::Param; }
2312     bool is_mvar() const { return kind() == level_kind::MVar; }
2313
2314     friend inline level const &max_lhs(level const &l) {
2315         lean_assert(l.is_max());
2316         return static_cast<level const &>(cnstr_get_ref(l, 0));
2317     }
2318     friend inline level const &max_rhs(level const &l) {
2319         lean_assert(l.is_max());
2320         return static_cast<level const &>(cnstr_get_ref(l, 1));
2321     }
2322     friend inline level const &imax_lhs(level const &l) {
2323         lean_assert(l.is_imax());
2324         return static_cast<level const &>(cnstr_get_ref(l, 0));
2325     }
2326     friend inline level const &imax_rhs(level const &l) {
2327         lean_assert(l.is_imax());
2328         return static_cast<level const &>(cnstr_get_ref(l, 1));
2329     }
2330     friend inline level const &level_lhs(level const &l) {
2331         lean_assert(l.is_max() || l.is_imax());
2332         return static_cast<level const &>(cnstr_get_ref(l, 0));
2333     }
2334     friend inline level const &level_rhs(level const &l) {
2335         lean_assert(l.is_max() || l.is_imax());
2336         return static_cast<level const &>(cnstr_get_ref(l, 1));
2337     }
2338     friend inline level const &succ_of(level const &l) {
2339         lean_assert(l.is_succ());
2340         return static_cast<level const &>(cnstr_get_ref(l, 0));
2341     }
2342     friend inline name const &param_id(level const &l) {
2343         lean_assert(l.is_param());
2344         return static_cast<name const &>(cnstr_get_ref(l, 0));
2345     }
2346     friend inline name const &mvar_id(level const &l) {
2347         lean_assert(l.is_mvar());
2348         return static_cast<name const &>(cnstr_get_ref(l, 0));
2349     }
2350     friend inline name const &level_id(level const &l) {
2351         lean_assert(l.is_param() || l.is_mvar());
2352         return static_cast<name const &>(cnstr_get_ref(l, 0));
2353     }
2354 };
2355
2356 typedef list_ref<level> levels;
2357 typedef pair<level, level> level_pair;
2358
2359 bool operator==(level const &l1, level const &l2);
2360 inline bool operator!=(level const &l1, level const &l2) {
2361     return !operator==(l1, l2);
2362 }
2363
2364 struct level_hash {
2365     unsigned operator()(level const &n) const { return n.hash(); }
2366 };
2367 struct level_eq {
2368     bool operator()(level const &n1, level const &n2) const { return n1 == n2; }
2369 };
2370
2371 inline serializer &operator<<(serializer &s, level const &l) {
2372     l.serialize(s);
2373     return s;
2374 }
2375 inline serializer &operator<<(serializer &s, levels const &ls) {
2376     ls.serialize(s);
2377     return s;
2378 }
2379 inline level read_level(deserializer &d) { return level::deserialize(d); }

```

```

2380 inline levels read_levels(deserializer &d) { return read_list_ref<level>(d); }
2381 inline deserializer &operator>>(deserializer &d, level &l) {
2382     l = read_level(d);
2383     return d;
2384 }
2385
2386 inline optional<level> none_level() { return optional<level>(); }
2387 inline optional<level> some_level(level const &e) { return optional<level>(e); }
2388 inline optional<level> some_level(level &&e) {
2389     return optional<level>(std::forward<level>(e));
2390 }
2391
2392 level const &mk_level_zero();
2393 level const &mk_level_one();
2394 level mk_max_core(level const &l1, level const &l2);
2395 level mk_imax_core(level const &l1, level const &l2);
2396 level mk_max(level const &l1, level const &l2);
2397 level mk_imax(level const &l1, level const &l2);
2398 level mk_succ(level const &l);
2399 level mk_univ_param(name const &n);
2400 level mk_univ_mvar(name const &n);
2401
2402 /** \brief Convert (succ^k l) into (l, k). If l is not a succ, then return (l,
2403 * 0) */
2404 pair<level, unsigned> to_offset(level l);
2405
2406 inline unsigned hash(level const &l) { return l.hash(); }
2407 inline level kind kind(level const &l) { return l.kind(); }
2408 inline bool is_zero(level const &l) { return l.is_zero(); }
2409 inline bool is_param(level const &l) { return l.is_param(); }
2410 inline bool is_mvar(level const &l) { return l.is_mvar(); }
2411 inline bool is_succ(level const &l) { return l.is_succ(); }
2412 inline bool is_max(level const &l) { return l.is_max(); }
2413 inline bool is_imax(level const &l) { return l.is_imax(); }
2414 bool is_one(level const &l);
2415
2416 unsigned get_depth(level const &l);
2417
2418 /** \brief Return true iff \c l is an explicit level.
2419     We say a level l is explicit iff
2420     1) l is zero OR
2421     2) l = succ(l') and l' is explicit */
2422 bool is_explicit(level const &l);
2423 /** \brief Convert an explicit universe into a unsigned integer.
2424     \pre is_explicit(l) */
2425 unsigned to_explicit(level const &l);
2426 /** \brief Return true iff \c l contains placeholder (aka meta parameters). */
2427 bool has_mvar(level const &l);
2428 /** \brief Return true iff \c l contains parameters */
2429 bool has_param(level const &l);
2430
2431 /** \brief Return a new level expression based on <tt>l == succ(arg)</tt>, where
2432     \c arg is replaced with \c new_arg. \pre is_succ(l) */
2433 level update_succ(level const &l, level const &new_arg);
2434 /** \brief Return a new level expression based on <tt>l == max(lhs, rhs)</tt>,
2435     where \c lhs is replaced with \c new_lhs and \c rhs is replaced with \c
2436     new_rhs.
2437     \pre is_max(l) || is_imax(l) */
2438 level update_max(level const &l, level const &new_lhs, level const &new_rhs);
2440
2441 /** \brief Return true if lhs and rhs denote the same level.
2442     The check is done by normalization. */
2443 bool is_equivalent(level const &lhs, level const &rhs);
2444 /** \brief Return the given level expression normal form */
2445 level normalize(level const &l);
2446
2447 /** \brief If the result is true, then forall assignments \c A that assigns all
2448     parameters and metavariables occurring in \c l1 and \c l2, we have that the
2449     universe level l1[A] is bigger or equal to l2[A].

```



```

2450
2451     \remark This function assumes l1 and l2 are normalized */
2452 bool is_geq_core(level l1, level l2);
2453
2454 bool is_geq(level const &l1, level const &l2);
2455
2456 bool levels_has_mvar(object *ls);
2457 bool has_mvar(levels const &ls);
2458 bool levels_has_param(object *ls);
2459 bool has_param(levels const &ls);
2460
2461 /** \brief An arbitrary (monotonic) total order on universe level terms. */
2462 bool is_lt(level const &l1, level const &l2, bool use_hash);
2463 bool is_lt(levels const &as, levels const &bs, bool use_hash);
2464 struct level_quick_cmp {
2465     int operator()(level const &l1, level const &l2) const {
2466         return is_lt(l1, l2, true) ? -1 : (l1 == l2 ? 0 : 1);
2467     }
2468 };
2469
2470 /** \brief Functional for applying <tt>F</tt> to each level expressions. */
2471 class for_each_level_fn {
2472     std::function<bool(level const &)> m_f; // NOLINT
2473     void apply(level const &l);
2474
2475     public:
2476     template <typename F>
2477     for_each_level_fn(F const &f) : m_f(f) {}
2478     void operator()(level const &l) { return apply(l); }
2479 };
2480 template <typename F>
2481 void for_each(level const &l, F const &f) {
2482     return for_each_level_fn(f)(l);
2483 }
2484
2485 /** \brief Functional for applying <tt>F</tt> to the level expressions. */
2486 class replace_level_fn {
2487     std::function<optional<level>(level const &)> m_f;
2488     level apply(level const &l);
2489
2490     public:
2491     template <typename F>
2492     replace_level_fn(F const &f) : m_f(f) {}
2493     level operator()(level const &l) { return apply(l); }
2494 };
2495 template <typename F>
2496 level replace(level const &l, F const &f) {
2497     return replace_level_fn(f)(l);
2498 }
2499
2500 /** \brief Return true if \c u occurs in \c l */
2501 bool occurs(level const &u, level const &l);
2502
2503 /** \brief If \c l contains a parameter that is not in \c ps, then return it.
2504  * Otherwise, return none. */
2505 optional<name> get_undef_param(level const &l, names const &lparams);
2506
2507 /** \brief Instantiate the universe level parameters \c ps occurring in \c l
2508     with the levels \c ls. \pre length(ps) == length(ls) */
2509 level instantiate(level const &l, names const &ps, levels const &ls);
2510
2511 /** \brief Printer for debugging purposes */
2512 std::ostream &operator<<(std::ostream &out, level const &l);
2513
2514 /** \brief If the result is true, then forall assignments \c A that assigns all
2515     parameters and metavariables occurring in \c l, l[A] != zero. */
2516 bool is_not_zero(level const &l);
2517
2518 /** \brief Pretty print the given level expression, unicode characters are used
2519     * if \c unicode is \c true. */

```



```

2520 format pp(level l, bool unicode, unsigned indent);
2521 /** \brief Pretty print the given level expression using the given configuration
2522 * options. */
2523 format pp(level const &l, options const &opts = options());
2524
2525 /** \brief Pretty print lhs <= rhs, unicode characters are used if \c unicode is
2526 * \c true. */
2527 format pp(level const &l, level const &r, bool unicode, unsigned indent);
2528 /** \brief Pretty print lhs <= rhs using the given configuration options. */
2529 format pp(level const &l, level const &r, options const &opts = options());
2530 /** \brief Convert a list of universe level parameter names into a list of
2531 * levels. */
2532 levels lparams_to_levels(names const &ps);
2533
2534 void initialize_level();
2535 void finalize_level();
2536 } // namespace lean
2537 void print(lean::level const &l);
2538 // ::::::::::::::::::::
2539 // local_ctx.h
2540 // ::::::::::::::::::::
2541 /*
2542 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
2543 Released under Apache 2.0 license as described in the file LICENSE.
2544
2545 Author: Leonardo de Moura
2546 */
2547 #pragma once
2548 #include "kernel/expr.h"
2549 #include "util/name_generator.h"
2550 #include "util/name_map.h"
2551 #include "util/rb_map.h"
2552
2553 namespace lean {
2554 /*
2555 inductive LocalDecl
2556 | cdecl (index : Nat) (name : Name) (userName : Name) (type : Expr) (bi :
2557 BinderInfo) | ldecl (index : Nat) (name : Name) (userName : Name) (type : Expr)
2558 (value : Expr)
2559 */
2560 class local_decl : public object_ref {
2561     friend class local_ctx;
2562     friend class local_context;
2563     friend void initialize_local_ctx();
2564     local_decl(unsigned idx, name const &n, name const &un, expr const &t,
2565               expr const &v);
2566     local_decl(local_decl const &d, expr const &t, expr const &v);
2567     local_decl(unsigned idx, name const &n, name const &un, expr const &t,
2568               binder_info bi);
2569     local_decl(local_decl const &d, expr const &t);
2570
2571 public:
2572     local_decl();
2573     local_decl(local_decl const &other) : object_ref(other) {}
2574     local_decl(local_decl &&other) : object_ref(other) {}
2575     local_decl(obj_arg o) : object_ref(o) {}
2576     local_decl(b_obj_arg o, bool) : object_ref(o, true) {}
2577     local_decl &operator=(local_decl const &other) {
2578         object_ref::operator=(other);
2579         return *this;
2580     }
2581     local_decl &operator=(local_decl &&other) {
2582         object_ref::operator=(other);
2583         return *this;
2584     }
2585     friend bool is_eqp(local_decl const &d1, local_decl const &d2) {
2586         return d1.raw() == d2.raw();
2587     }
2588     unsigned get_idx() const {
2589         return static_cast<nat const &>(cnstr_get_ref(raw(), 0))

```

```

2590         .get_small_value();
2591     }
2592     name const &get_name() const {
2593         return static_cast<name const &>(cnstr_get_ref(raw(), 1));
2594     }
2595     name const &get_user_name() const {
2596         return static_cast<name const &>(cnstr_get_ref(raw(), 2));
2597     }
2598     expr const &get_type() const {
2599         return static_cast<expr const &>(cnstr_get_ref(raw(), 3));
2600     }
2601     optional<expr> get_value() const {
2602         if (cnstr_tag(raw()) == 0) return none_expr();
2603         return some_expr(static_cast<expr const &>(cnstr_get_ref(raw(), 4)));
2604     }
2605     binder_info get_info() const;
2606     expr mk_ref() const;
2607 };
2608
2609 /* Plain local context object used by the kernel type checker. */
2610 class local_ctx : public object_ref {
2611     protected:
2612         template <bool is_lambda>
2613         expr mk_binding(unsigned num, expr const *fvars, expr const &b,
2614             bool remove_dead_let = false) const;
2615
2616     public:
2617         local_ctx();
2618         explicit local_ctx(obj_arg o) : object_ref(o) {}
2619         local_ctx(b_obj_arg o, bool) : object_ref(o, true) {}
2620         local_ctx(local_ctx const &other) : object_ref(other) {}
2621         local_ctx(local_ctx &&other) : object_ref(other) {}
2622         local_ctx &operator=(local_ctx const &other) {
2623             object_ref::operator=(other);
2624             return *this;
2625         }
2626         local_ctx &operator=(local_ctx &&other) {
2627             object_ref::operator=(other);
2628             return *this;
2629         }
2630
2631         bool empty() const;
2632
2633         /* Low level `mk_local_decl` */
2634         local_decl mk_local_decl(name const &n, name const &un, expr const &type,
2635             binder_info bi);
2636         /* Low level `mk_local_decl` */
2637         local_decl mk_local_decl(name const &n, name const &un, expr const &type,
2638             expr const &value);
2639
2640         expr mk_local_decl(name_generator &g, name const &un, expr const &type,
2641             binder_info bi = mk_binder_info()) {
2642             return mk_local_decl(g.next(), un, type, bi).mk_ref();
2643         }
2644
2645         expr mk_local_decl(name_generator &g, name const &un, expr const &type,
2646             expr const &value) {
2647             return mk_local_decl(g.next(), un, type, value).mk_ref();
2648         }
2649
2650         /** \brief Return the local declarations for the given reference. */
2651         optional<local_decl> find_local_decl(name const &n) const;
2652         optional<local_decl> find_local_decl(expr const &e) const {
2653             return find_local_decl(fvar_name(e));
2654         }
2655
2656         local_decl get_local_decl(name const &n) const;
2657         local_decl get_local_decl(expr const &e) const {
2658             return get_local_decl(fvar_name(e));
2659         }

```

```

2660
2661 /* \brief Return type of the given free variable.
2662 \pre is_fvar(e) */
2663 expr get_type(expr const &e) const { return get_local_decl(e).get_type(); }
2664
2665 /** Return the free variable associated with the given name.
2666 \pre get_local_decl(n) */
2667 expr get_local(name const &n) const;
2668
2669 /** \brief Remove the given local decl. */
2670 void clear(local_decl const &d);
2671
2672 expr mk_lambda(unsigned num, expr const *fvars, expr const &e,
2673               bool remove_dead_let = false) const;
2674 expr mk_pi(unsigned num, expr const *fvars, expr const &e,
2675            bool remove_dead_let = false) const;
2676 expr mk_lambda(buffer<expr> const &fvars, expr const &e,
2677               bool remove_dead_let = false) const {
2678     return mk_lambda(fvars.size(), fvars.data(), e, remove_dead_let);
2679 }
2680 expr mk_pi(buffer<expr> const &fvars, expr const &e,
2681            bool remove_dead_let = false) const {
2682     return mk_pi(fvars.size(), fvars.data(), e, remove_dead_let);
2683 }
2684 expr mk_lambda(expr const &fvar, expr const &e) {
2685     return mk_lambda(1, &fvar, e);
2686 }
2687 expr mk_pi(expr const &fvar, expr const &e) { return mk_pi(1, &fvar, e); }
2688 expr mk_lambda(std::initializer_list<expr> const &fvars, expr const &e) {
2689     return mk_lambda(fvars.size(), fvars.begin(), e);
2690 }
2691 expr mk_pi(std::initializer_list<expr> const &fvars, expr const &e) {
2692     return mk_pi(fvars.size(), fvars.begin(), e);
2693 }
2694 };
2695
2696 void initialize_local_ctx();
2697 void finalize_local_ctx();
2698 } // namespace lean
2699 // ::::::::::::::
2700 // quot.h
2701 // ::::::::::::::
2702 /*
2703 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
2704 Released under Apache 2.0 license as described in the file LICENSE.
2705
2706 Author: Leonardo de Moura
2707
2708 Quotient types.
2709 */
2710 #pragma once
2711 #include "kernel/environment.h"
2712
2713 namespace lean {
2714 class quot_consts {
2715     static name *g_quot;
2716     static name *g_quot_lift;
2717     static name *g_quot_ind;
2718     static name *g_quot_mk;
2719
2720     friend bool quot_is_decl(name const &n);
2721     friend bool quot_is_rec(name const &n);
2722     template <typename WHNF>
2723     friend optional<expr> quot_reduce_rec(expr const &e, WHNF const &whnf);
2724     template <typename WHNF, typename IS_STUCK>
2725     friend optional<expr> quot_is_stuck(expr const &e, WHNF const &whnf,
2726                                       IS_STUCK const &is_stuck);
2727     friend class environment;
2728     friend void initialize_quot();
2729     friend void finalize_quot();

```

```

2730 };
2731
2732 inline bool quot_is_decl(name const &n) {
2733     return n == *quot_consts::g_quot || n == *quot_consts::g_quot_lift ||
2734            n == *quot_consts::g_quot_ind || n == *quot_consts::g_quot_mk;
2735 }
2736
2737 inline bool quot_is_rec(name const &n) {
2738     return n == *quot_consts::g_quot_lift || n == *quot_consts::g_quot_ind;
2739 }
2740
2741 /** \brief Try to reduce a `quot` recursor application (i.e., `quot.lift` or
2742     `quot.ind` application).
2743
2744     `whnf : expr -> expr` */
2745 template <typename WHNF>
2746 optional<expr> quot_reduce_rec(expr const &e, WHNF const &whnf) {
2747     expr const &fn = get_app_fn(e);
2748     if (!is_constant(fn)) return none_expr();
2749     unsigned mk_pos;
2750     unsigned arg_pos;
2751     if (const_name(fn) == *quot_consts::g_quot_lift) {
2752         mk_pos = 5;
2753         arg_pos = 3;
2754     } else if (const_name(fn) == *quot_consts::g_quot_ind) {
2755         mk_pos = 4;
2756         arg_pos = 3;
2757     } else {
2758         return none_expr();
2759     }
2760     buffer<expr> args;
2761     get_app_args(e, args);
2762     if (args.size() <= mk_pos) return none_expr();
2763
2764     expr mk = whnf(args[mk_pos]);
2765     expr const &mk_fn = get_app_fn(mk);
2766     if (!is_constant(mk_fn) || const_name(mk_fn) != *quot_consts::g_quot_mk)
2767         return none_expr();
2768
2769     expr const &f = args[arg_pos];
2770     expr r = mk_app(f, app_arg(mk));
2771     unsigned elim_arity = mk_pos + 1;
2772     if (args.size() > elim_arity)
2773         r = mk_app(r, args.size() - elim_arity, args.begin() + elim_arity);
2774     return some_expr(r);
2775 }
2776
2777 /** \brief Return a non-none expression that is preventing the `quot` recursor
2778     application from being reduced.
2779
2780     `whnf : expr -> expr`
2781     `is_stuck : expr -> optional<expr>` */
2782 template <typename WHNF, typename IS_STUCK>
2783 optional<expr> quot_is_stuck(expr const &e, WHNF const &whnf,
2784                             IS_STUCK const &is_stuck) {
2785     expr const &fn = get_app_fn(e);
2786     if (!is_constant(fn)) return none_expr();
2787     unsigned mk_pos;
2788     if (const_name(fn) == *quot_consts::g_quot_lift) {
2789         mk_pos = 5;
2790     } else if (const_name(fn) == *quot_consts::g_quot_ind) {
2791         mk_pos = 4;
2792     } else {
2793         return none_expr();
2794     }
2795
2796     buffer<expr> args;
2797     get_app_args(e, args);
2798     if (args.size() <= mk_pos) return none_expr();
2799

```

```

2800     return is_stuck(whnf(args[mk_pos]));
2801 }
2802
2803 void initialize_quot();
2804 void finalize_quot();
2805 } // namespace lean
2806 // ::::::::::::::
2807 // replace_fn.h
2808 // ::::::::::::::
2809 /*
2810 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
2811 Released under Apache 2.0 license as described in the file LICENSE.
2812
2813 Author: Leonardo de Moura
2814 */
2815 #pragma once
2816 #include <lean/interrupt.h>
2817
2818 #include <tuple>
2819
2820 #include "kernel/expr.h"
2821 #include "kernel/expr_maps.h"
2822 #include "util/buffer.h"
2823
2824 namespace lean {
2825 /**
2826  \brief Apply <tt>f</tt> to the subexpressions of a given expression.
2827
2828  f is invoked for each subexpression \c s of the input expression e.
2829  In a call <tt>f(s, n)</tt>, n is the scope level, i.e., the number of
2830  bindings operators that enclosing \c s. The replaces only visits children of
2831  \c e if f return none_expr.
2832 */
2833 expr replace(expr const &e,
2834             std::function<optional<expr>(expr const &, unsigned)> const &f,
2835             bool use_cache = true);
2836 inline expr replace(expr const &e,
2837                   std::function<optional<expr>(expr const &)> const &f,
2838                   bool use_cache = true) {
2839     return replace(
2840         e, [&](expr const &e, unsigned) { return f(e); }, use_cache);
2841 }
2842 } // namespace lean
2843 // ::::::::::::::
2844 // type_checker.h
2845 // ::::::::::::::
2846 /*
2847 Copyright (c) 2013-14 Microsoft Corporation. All rights reserved.
2848 Released under Apache 2.0 license as described in the file LICENSE.
2849
2850 Author: Leonardo de Moura
2851 */
2852 #pragma once
2853 #include <algorithm>
2854 #include <memory>
2855 #include <unordered_set>
2856 #include <utility>
2857
2858 #include "kernel/environment.h"
2859 #include "kernel/equiv_manager.h"
2860 #include "kernel/expr_maps.h"
2861 #include "kernel/local_ctx.h"
2862 #include "util/lbool.h"
2863 #include "util/name_generator.h"
2864 #include "util/name_set.h"
2865
2866 namespace lean {
2867 /** \brief Lean Type Checker. It can also be used to infer types, check whether
2868     a type \c A is convertible to a type \c B, etc. */
2869 class type_checker {

```

```

2870 public:
2871     class state {
2872         typedef expr_map<expr> infer_cache;
2873         typedef std::unordered_set<expr_pair, expr_pair_hash, expr_pair_eq>
2874             expr_pair_set;
2875         environment m_env;
2876         name_generator m_ngen;
2877         infer_cache m_infer_type[2];
2878         expr_map<expr> m_wnhf_core;
2879         expr_map<expr> m_wnhf;
2880         equiv_manager m_eqv_manager;
2881         expr_pair_set m_failure;
2882         friend type_checker;
2883
2884     public:
2885         state(environment const &env);
2886         environment &env() { return m_env; }
2887         environment const &env() const { return m_env; }
2888         name_generator &ngen() { return m_ngen; }
2889     };
2890
2891 private:
2892     bool m_st_owner;
2893     state *m_st;
2894     local_ctx m_lctx;
2895     bool m_safe_only;
2896     /* When `m_lparams != nullptr, the `check` method makes sure all level
2897        parameters are in `m_lparams`. */
2898     names const *m_lparams;
2899
2900     expr ensure_sort_core(expr e, expr const &s);
2901     expr ensure_pi_core(expr e, expr const &s);
2902     void check_level(level const &l);
2903     expr infer_fvar(expr const &e);
2904     expr infer_constant(expr const &e, bool infer_only);
2905     expr infer_lambda(expr const &e, bool infer_only);
2906     expr infer_pi(expr const &e, bool infer_only);
2907     expr infer_app(expr const &e, bool infer_only);
2908     expr infer_proj(expr const &e, bool infer_only);
2909     expr infer_let(expr const &e, bool infer_only);
2910     expr infer_type_core(expr const &e, bool infer_only);
2911     expr infer_type(expr const &e);
2912
2913     enum class reduction_status { Continue, DefUnknown, DefEqual, DefDiff };
2914     optional<expr> reduce_recurser(expr const &e, bool cheap);
2915     optional<expr> reduce_proj(expr const &e, bool cheap);
2916     expr whnf_fvar(expr const &e, bool cheap);
2917     optional<constant_info> is_delta(expr const &e) const;
2918     optional<expr> unfold_definition_core(expr const &e);
2919
2920     bool is_def_eq_binding(expr t, expr s);
2921     bool is_def_eq(level const &l1, level const &l2);
2922     bool is_def_eq(levels const &ls1, levels const &ls2);
2923     lbool quick_is_def_eq(expr const &t, expr const &s, bool use_hash = false);
2924     lbool is_def_eq_offset(expr const &t, expr const &s);
2925     bool is_def_eq_args(expr t, expr s);
2926     bool try_eta_expansion_core(expr const &t, expr const &s);
2927     bool try_eta_expansion(expr const &t, expr const &s) {
2928         return try_eta_expansion_core(t, s) || try_eta_expansion_core(s, t);
2929     }
2930     lbool try_string_lit_expansion_core(expr const &t, expr const &s);
2931     lbool try_string_lit_expansion(expr const &t, expr const &s);
2932     bool is_def_eq_app(expr const &t, expr const &s);
2933     bool is_def_eq_proof_irrel(expr const &t, expr const &s);
2934     bool failed_before(expr const &t, expr const &s) const;
2935     void cache_failure(expr const &t, expr const &s);
2936     reduction_status lazy_delta_reduction_step(expr &t_n, expr &s_n);
2937     lbool lazy_delta_reduction(expr &t_n, expr &s_n);
2938     bool is_def_eq_core(expr const &t, expr const &s);
2939     /** \brief Like \c check, but ignores undefined universes */

```



```

2940     expr check_ignore_undefined_universes(expr const &e);
2941
2942     template <typename F>
2943     optional<expr> reduce_bin_nat_op(F const &f, expr const &e);
2944     template <typename F>
2945     optional<expr> reduce_bin_nat_pred(F const &f, expr const &e);
2946     optional<expr> reduce_nat(expr const &e);
2947
2948 public:
2949     type_checker(state &st, local_ctx const &lctx, bool safe_only = true);
2950     type_checker(state &st, bool safe_only = true)
2951         : type_checker(st, local_ctx(), safe_only) {}
2952     type_checker(environment const &env, local_ctx const &lctx,
2953                 bool safe_only = true);
2954     type_checker(environment const &env, bool safe_only = true)
2955         : type_checker(env, local_ctx(), safe_only) {}
2956     type_checker(type_checker &&);
2957     type_checker(type_checker const &) = delete;
2958     ~type_checker();
2959
2960     environment const &env() const { return m_st->m_env; }
2961
2962     /** \brief Return the type of \c t.
2963         It does not check whether the input expression is type correct or not.
2964         The contract is: IF the input expression is type correct, then the
2965         inferred type is correct. Throw an exception if a type error is found. */
2966     expr infer(expr const &t) { return infer_type(t); }
2967
2968     /** \brief Type check the given expression, and return the type of \c t.
2969         Throw an exception if a type error is found. */
2970     expr check(expr const &t, names const &ps);
2971     /** \brief Like \c check, but ignores undefined universes */
2972     expr check(expr const &t) { return check_ignore_undefined_universes(t); }
2973
2974     /** \brief Return true iff t is definitionally equal to s. */
2975     bool is_def_eq(expr const &t, expr const &s);
2976     /** \brief Return true iff t is a proposition. */
2977     bool is_prop(expr const &t);
2978     /** \brief Return the weak head normal form of \c t. */
2979     expr whnf(expr const &t);
2980     /** \brief Return a Pi if \c t is convertible to a Pi type. Throw an
2981         exception otherwise. The argument \c s is used when reporting errors */
2982     expr ensure_pi(expr const &t, expr const &s);
2983     expr ensure_pi(expr const &t) { return ensure_pi(t, t); }
2984     /** \brief Make sure type of \c e is a Pi, and return it. Throw an exception
2985         * otherwise. */
2986     expr ensure_fun(expr const &e) { return ensure_pi(infer(e), e); }
2987     /** \brief Return a Sort if \c t is convertible to Sort. Throw an exception
2988         otherwise. The argument \c s is used when reporting errors. */
2989     expr ensure_sort(expr const &t, expr const &s);
2990     /** \brief Return a Sort if \c t is convertible to Sort. Throw an exception
2991         * otherwise. */
2992     expr ensure_sort(expr const &t) { return ensure_sort(t, t); }
2993     /** \brief Make sure type of \c e is a sort, and return it. Throw an
2994         exception otherwise. */
2995     expr ensure_type(expr const &e) { return ensure_sort(infer(e), e); }
2996     expr eta_expand(expr const &e);
2997
2998     expr whnf_core(expr const &e, bool cheap = false);
2999     optional<expr> unfold_definition(expr const &e);
3000 };
3001
3002 void initialize_type_checker();
3003 void finalize_type_checker();
3004 } // namespace lean
3005 // ::::::::::::::
3006 // abstract.cpp
3007 // ::::::::::::::
3008 /*
3009 Copyright (c) 2013 Microsoft Corporation. All rights reserved.

```



```

3010 Released under Apache 2.0 license as described in the file LICENSE.
3011
3012 Author: Leonardo de Moura
3013 */
3014 #include <algorithm>
3015 #include <utility>
3016 #include <vector>
3017
3018 #include "kernel/abstract.h"
3019 #include "kernel/replace_fn.h"
3020
3021 namespace lean {
3022 expr abstract(expr const &e, unsigned n, expr const *subst) {
3023     lean_assert(std::all_of(subst, subst + n, [](expr const &e) {
3024         return !has_loose_bvars(e) && is_fvar(e);
3025     }));
3026     if (!has_fvar(e)) return e;
3027     return replace(e, [](expr const &m, unsigned offset) -> optional<expr> {
3028         if (!has_fvar(m))
3029             return some_expr(
3030                 m); // expression m does not contain free variables
3031         if (is_fvar(m)) {
3032             unsigned i = n;
3033             while (i > 0) {
3034                 --i;
3035                 if (fvar_name(subst[i]) == fvar_name(m))
3036                     return some_expr(mk_bvar(offset + n - i - 1));
3037             }
3038             return none_expr();
3039         }
3040         return none_expr();
3041     });
3042 }
3043
3044 expr abstract(expr const &e, name const &n) {
3045     expr fvar = mk_fvar(n);
3046     return abstract(e, 1, &fvar);
3047 }
3048
3049 static object *lean_expr_abstract_core(object *e0, size_t n, object *subst) {
3050     lean_assert(n <= lean_array_size(subst));
3051     expr const &e = reinterpret_cast<expr const &>(e0);
3052     if (!has_fvar(e)) {
3053         lean_inc(e0);
3054         return e0;
3055     }
3056     expr r = replace(e, [](expr const &m, unsigned offset) -> optional<expr> {
3057         if (!has_fvar(m))
3058             return some_expr(
3059                 m); // expression m does not contain free variables
3060         if (is_fvar(m)) {
3061             size_t i = n;
3062             while (i > 0) {
3063                 --i;
3064                 object *v = lean_array_get_core(subst, i);
3065                 if (is_fvar_core(v) && fvar_name_core(v) == fvar_name(m))
3066                     return some_expr(mk_bvar(offset + n - i - 1));
3067             }
3068             return none_expr();
3069         }
3070         return none_expr();
3071     });
3072     return r.steal();
3073 }
3074
3075 extern "C" object *lean_expr_abstract_range(object *e, object *n,
3076                                             object *subst) {
3077     if (!lean_is_scalar(n))
3078         return lean_expr_abstract_core(e, lean_array_size(subst), subst);
3079     else

```

```

3080         return lean_expr_abstract_core(
3081             e, std::min(lean_unbox(n), lean_array_size(subst)), subst);
3082     }
3083
3084     extern "C" object *lean_expr_abstract(object *e, object *subst) {
3085         return lean_expr_abstract_core(e, lean_array_size(subst), subst);
3086     }
3087 } // namespace lean
3088 // ::::::::::::::
3089 // declaration.cpp
3090 // ::::::::::::::
3091 /*
3092 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
3093 Released under Apache 2.0 license as described in the file LICENSE.
3094
3095 Author: Leonardo de Moura
3096 */
3097 #include "kernel/declaration.h"
3098 #include "kernel/environment.h"
3099 #include "kernel/for_each_fn.h"
3100
3101 namespace lean {
3102
3103     extern "C" object *lean_mk_reducibility_hints_regular(uint32 h);
3104     extern "C" uint32 lean_reducibility_hints_get_height(object *o);
3105
3106     reducibility_hints reducibility_hints::mk_regular(unsigned h) {
3107         return reducibility_hints(lean_mk_reducibility_hints_regular(h));
3108     }
3109
3110     unsigned reducibility_hints::get_height() const {
3111         return lean_reducibility_hints_get_height(to_obj_arg());
3112     }
3113
3114     int compare(reducibility_hints const &h1, reducibility_hints const &h2) {
3115         if (h1.kind() == h2.kind()) {
3116             if (h1.kind() == reducibility_hints_kind::Regular) {
3117                 if (h1.get_height() == h2.get_height())
3118                     return 0; /* unfold both */
3119                 else if (h1.get_height() > h2.get_height())
3120                     return -1; /* unfold f1 */
3121                 else
3122                     return 1; /* unfold f2 */
3123                 return h1.get_height() > h2.get_height() ? -1 : 1;
3124             } else {
3125                 return 0; /* reduce both */
3126             }
3127         } else {
3128             if (h1.kind() == reducibility_hints_kind::Opaque) {
3129                 return 1; /* reduce f2 */
3130             } else if (h2.kind() == reducibility_hints_kind::Opaque) {
3131                 return -1; /* reduce f1 */
3132             } else if (h1.kind() == reducibility_hints_kind::Abbreviation) {
3133                 return -1; /* reduce f1 */
3134             } else if (h2.kind() == reducibility_hints_kind::Abbreviation) {
3135                 return 1; /* reduce f2 */
3136             } else {
3137                 lean_unreachable();
3138             }
3139         }
3140     }
3141
3142     constant_val::constant_val(name const &n, names const &lparams,
3143                                expr const &type)
3144         : object_ref(mk_cnstr(0, n, lparams, type)) {}
3145
3146     extern "C" object *lean_mk_axiom_val(object *n, object *lparams, object *type,
3147                                         uint8 is_unsafe);
3148     extern "C" uint8 lean_axiom_val_is_unsafe(object *v);
3149

```

```

3150 axiom_val::axiom_val(name const &n, names const &lparams, expr const &type,
3151                        bool is_unsafe)
3152   : object_ref(lean_mk_axiom_val(n.to_obj_arg(), lparams.to_obj_arg(),
3153                                  type.to_obj_arg(), is_unsafe)) {}
3154
3155 bool axiom_val::is_unsafe() const {
3156   return lean_axiom_val_is_unsafe(to_obj_arg());
3157 }
3158
3159 extern "C" object *lean_mk_definition_val(object *n, object *lparams,
3160                                           object *type, object *value,
3161                                           object *hints, uint8 safety);
3162 extern "C" uint8 lean_definition_val_get_safety(object *v);
3163
3164 definition_val::definition_val(name const &n, names const &lparams,
3165                               expr const &type, expr const &val,
3166                               reducibility_hints const &hints,
3167                               definition_safety safety)
3168   : object_ref(lean_mk_definition_val(
3169       n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
3170       val.to_obj_arg(), hints.to_obj_arg(), static_cast<uint8>(safety))) {}
3171
3172 definition_safety definition_val::get_safety() const {
3173   return static_cast<definition_safety>(
3174       lean_definition_val_get_safety(to_obj_arg()));
3175 }
3176
3177 theorem_val::theorem_val(name const &n, names const &lparams, expr const &type,
3178                          expr const &val)
3179   : object_ref(mk_cnstr(0, constant_val(n, lparams, type), val)) {}
3180
3181 extern "C" object *lean_mk_opaque_val(object *n, object *lparams, object *type,
3182                                       object *value, uint8 is_unsafe);
3183 extern "C" uint8 lean_opaque_val_is_unsafe(object *v);
3184
3185 opaque_val::opaque_val(name const &n, names const &lparams, expr const &type,
3186                        expr const &val, bool is_unsafe)
3187   : object_ref(lean_mk_opaque_val(n.to_obj_arg(), lparams.to_obj_arg(),
3188                                   type.to_obj_arg(), val.to_obj_arg(),
3189                                   is_unsafe)) {}
3190
3191 bool opaque_val::is_unsafe() const {
3192   return lean_opaque_val_is_unsafe(to_obj_arg());
3193 }
3194
3195 extern "C" object *lean_mk_quot_val(object *n, object *lparams, object *type,
3196                                     uint8 k);
3197 extern "C" uint8 lean_quot_val_kind(object *v);
3198
3199 quot_val::quot_val(name const &n, names const &lparams, expr const &type,
3200                   quot_kind k)
3201   : object_ref(lean_mk_quot_val(n.to_obj_arg(), lparams.to_obj_arg(),
3202                                 type.to_obj_arg(), static_cast<uint8>(k))) {}
3203
3204 quot_kind quot_val::get_quot_kind() const {
3205   return static_cast<quot_kind>(lean_quot_val_kind(to_obj_arg()));
3206 }
3207
3208 recursor_rule::recursor_rule(name const &cnstr, unsigned nfields,
3209                              expr const &rhs)
3210   : object_ref(mk_cnstr(0, cnstr, nat(nfields), rhs)) {}
3211
3212 extern "C" object *lean_mk_inductive_val(object *n, object *lparams,
3213                                          object *type, object *nparams,
3214                                          object *nindices, object *all,
3215                                          object *cnstrs, uint8 rec,
3216                                          uint8 unsafe, uint8 is_refl,
3217                                          uint8 is_nested);
3218 extern "C" uint8 lean_inductive_val_is_rec(object *v);
3219 extern "C" uint8 lean_inductive_val_is_unsafe(object *v);

```

```

3220 extern "C" uint8 lean_inductive_val_is_reflexive(object *v);
3221 extern "C" uint8 lean_inductive_val_is_nested(object *v);
3222
3223 inductive_val::inductive_val(name const &n, names const &lparams,
3224                               expr const &type, unsigned nparams,
3225                               unsigned nindices, names const &all,
3226                               names const &cnstrs, bool rec, bool unsafe,
3227                               bool is_refl, bool is_nested)
3228   : object_ref(lean_mk_inductive_val(
3229       n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
3230       nat(nparams).to_obj_arg(), nat(nindices).to_obj_arg(),
3231       all.to_obj_arg(), cnstrs.to_obj_arg(), rec, unsafe, is_refl,
3232       is_nested)) {}
3233
3234 bool inductive_val::is_rec() const {
3235     return lean_inductive_val_is_rec(to_obj_arg());
3236 }
3237 bool inductive_val::is_unsafe() const {
3238     return lean_inductive_val_is_unsafe(to_obj_arg());
3239 }
3240 bool inductive_val::is_reflexive() const {
3241     return lean_inductive_val_is_reflexive(to_obj_arg());
3242 }
3243 bool inductive_val::is_nested() const {
3244     return lean_inductive_val_is_nested(to_obj_arg());
3245 }
3246
3247 extern "C" object *lean_mk_constructor_val(object *n, object *lparams,
3248                                           object *type, object *induct,
3249                                           object *cidx, object *nparams,
3250                                           object *nfields, uint8 unsafe);
3251 extern "C" uint8 lean_constructor_val_is_unsafe(object *v);
3252
3253 constructor_val::constructor_val(name const &n, names const &lparams,
3254                                  expr const &type, name const &induct,
3255                                  unsigned cidx, unsigned nparams,
3256                                  unsigned nfields, bool is_unsafe)
3257   : object_ref(lean_mk_constructor_val(
3258       n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
3259       induct.to_obj_arg(), nat(cidx).to_obj_arg(),
3260       nat(nparams).to_obj_arg(), nat(nfields).to_obj_arg(), is_unsafe)) {}
3261
3262 bool constructor_val::is_unsafe() const {
3263     return lean_constructor_val_is_unsafe(to_obj_arg());
3264 }
3265
3266 extern "C" object *lean_mk_recursor_val(object *n, object *lparams,
3267                                         object *type, object *all,
3268                                         object *nparams, object *nindices,
3269                                         object *nmotives, object *nminors,
3270                                         object *rules, uint8 k, uint8 unsafe);
3271 extern "C" uint8 lean_recursor_k(object *v);
3272 extern "C" uint8 lean_recursor_is_unsafe(object *v);
3273
3274 recursor_val::recursor_val(name const &n, names const &lparams,
3275                             expr const &type, names const &all, unsigned nparams,
3276                             unsigned nindices, unsigned nmotives,
3277                             unsigned nminors, recursor_rules const &rules,
3278                             bool k, bool is_unsafe)
3279   : object_ref(lean_mk_recursor_val(
3280       n.to_obj_arg(), lparams.to_obj_arg(), type.to_obj_arg(),
3281       all.to_obj_arg(), nat(nparams).to_obj_arg(),
3282       nat(nindices).to_obj_arg(), nat(nmotives).to_obj_arg(),
3283       nat(nminors).to_obj_arg(), rules.to_obj_arg(), k, is_unsafe)) {}
3284
3285 bool recursor_val::is_k() const { return lean_recursor_k(to_obj_arg()); }
3286 bool recursor_val::is_unsafe() const {
3287     return lean_recursor_is_unsafe(to_obj_arg());
3288 }
3289

```

```

3290 bool declaration::is_unsafe() const {
3291     switch (kind()) {
3292         case declaration_kind::Definition:
3293             return to_definition_val().get_safety() ==
3294                 definition_safety::unsafe;
3295         case declaration_kind::Axiom:
3296             return to_axiom_val().is_unsafe();
3297         case declaration_kind::Theorem:
3298             return false;
3299         case declaration_kind::Opaque:
3300             return to_opaque_val().is_unsafe();
3301         case declaration_kind::Inductive:
3302             return inductive_decl(*this).is_unsafe();
3303         case declaration_kind::Quot:
3304             return false;
3305         case declaration_kind::MutualDefinition:
3306             return true;
3307     }
3308     lean_unreachable();
3309 }
3310
3311 bool use_unsafe(environment const &env, expr const &e) {
3312     bool found = false;
3313     for_each(e, [&](expr const &e, unsigned) {
3314         if (found) return false;
3315         if (is_constant(e)) {
3316             if (auto info = env.find(const_name(e))) {
3317                 if (info->is_unsafe()) {
3318                     found = true;
3319                     return false;
3320                 }
3321             }
3322         }
3323     });
3324     return found;
3325 }
3326
3327 static declaration *g_dummy = nullptr;
3328 declaration::declaration() : declaration(*g_dummy) {}
3329
3330 static unsigned get_max_height(environment const &env, expr const &v) {
3331     unsigned h = 0;
3332     for_each(v, [&](expr const &e, unsigned) {
3333         if (is_constant(e)) {
3334             auto d = env.find(const_name(e));
3335             if (d && d->get_hints().get_height() > h)
3336                 h = d->get_hints().get_height();
3337         }
3338     });
3339     return h;
3340 }
3341
3342 definition_val mk_definition_val(environment const &env, name const &n,
3343                                 names const &params, expr const &t,
3344                                 expr const &v, definition_safety s) {
3345     unsigned h = get_max_height(env, v);
3346     return definition_val(n, params, t, v,
3347                          reducibility_hints::mk_regular(h + 1), s);
3348 }
3349
3350 declaration mk_definition(name const &n, names const &params, expr const &t,
3351                           expr const &v, reducibility_hints const &h,
3352                           definition_safety safety) {
3353     return declaration(
3354         mk_cnstr(static_cast<unsigned>(declaration_kind::Definition),
3355                 definition_val(n, params, t, v, h, safety)));
3356 }
3357
3358 }
3359

```

```

3360 declaration mk_definition(environment const &env, name const &n,
3361                             names const &params, expr const &t, expr const &v,
3362                             definition_safety safety) {
3363     return declaration(
3364         mk_cnstr(static_cast<unsigned>(declaration_kind::Definition),
3365                 mk_definition_val(env, n, params, t, v, safety)));
3366 }
3367
3368 declaration mk_opaque(name const &n, names const &params, expr const &t,
3369                       expr const &v, bool is_unsafe) {
3370     return declaration(mk_cnstr(static_cast<unsigned>(declaration_kind::Opaque),
3371                                opaque_val(n, params, t, v, is_unsafe)));
3372 }
3373
3374 declaration mk_axiom(name const &n, names const &params, expr const &t,
3375                     bool unsafe) {
3376     return declaration(mk_cnstr(static_cast<unsigned>(declaration_kind::Axiom),
3377                                axiom_val(n, params, t, unsafe)));
3378 }
3379
3380 static definition_safety to_safety(bool unsafe) {
3381     return unsafe ? definition_safety::unsafe : definition_safety::safe;
3382 }
3383
3384 declaration mk_definition_inferring_unsafe(environment const &env,
3385                                             name const &n, names const &params,
3386                                             expr const &t, expr const &v,
3387                                             reducibility_hints const &hints) {
3388     bool unsafe = use_unsafe(env, t) || use_unsafe(env, v);
3389     return mk_definition(n, params, t, v, hints, to_safety(unsafe));
3390 }
3391
3392 declaration mk_definition_inferring_unsafe(environment const &env,
3393                                             name const &n, names const &params,
3394                                             expr const &t, expr const &v) {
3395     bool unsafe = use_unsafe(env, t) && use_unsafe(env, v);
3396     unsigned h = get_max_height(env, v);
3397     return mk_definition(n, params, t, v, reducibility_hints::mk_regular(h + 1),
3398                         to_safety(unsafe));
3399 }
3400
3401 inductive_type::inductive_type(name const &id, expr const &type,
3402                                constructors const &cnstrs)
3403     : object_ref(mk_cnstr(0, id, type, cnstrs)) {}
3404
3405 extern "C" object *lean_mk_inductive_decl(object *lparams, object *nparams,
3406                                           object *types, uint8 unsafe);
3407 extern "C" uint8 lean_is_unsafe_inductive_decl(object *d);
3408
3409 declaration mk_inductive_decl(names const &lparams, nat const &nparams,
3410                              inductive_types const &types, bool is_unsafe) {
3411     return declaration(lean_mk_inductive_decl(lparams.to_obj_arg(),
3412                                              nparams.to_obj_arg(),
3413                                              types.to_obj_arg(), is_unsafe));
3414 }
3415
3416 bool inductive_decl::is_unsafe() const {
3417     return lean_is_unsafe_inductive_decl(to_obj_arg());
3418 }
3419
3420 // =====
3421 // Constant info
3422 constant_info::constant_info() : constant_info(*g_dummy) {}
3423
3424 constant_info::constant_info(declaration const &d) : object_ref(d.raw()) {
3425     lean_assert(d.is_definition() || d.is_theorem() || d.is_axiom() ||
3426               d.is_opaque());
3427     inc_ref(d.raw());
3428 }
3429

```

```

3430 constant_info::constant_info(definition_val const &v)
3431     : object_ref(
3432         mk_cnstr(static_cast<unsigned>(constant_info_kind::Definition), v)) {}
3433
3434 constant_info::constant_info(quot_val const &v)
3435     : object_ref(mk_cnstr(static_cast<unsigned>(constant_info_kind::Quot), v)) {
3436 }
3437
3438 constant_info::constant_info(inductive_val const &v)
3439     : object_ref(
3440         mk_cnstr(static_cast<unsigned>(constant_info_kind::Inductive), v)) {}
3441
3442 constant_info::constant_info(constructor_val const &v)
3443     : object_ref(mk_cnstr(
3444         static_cast<unsigned>(constant_info_kind::Constructor), v)) {}
3445
3446 constant_info::constant_info(recursor_val const &v)
3447     : object_ref(
3448         mk_cnstr(static_cast<unsigned>(constant_info_kind::Recursor), v)) {}
3449
3450 static reducibility_hints *g_opaque = nullptr;
3451
3452 reducibility_hints const &constant_info::get_hints() const {
3453     if (is_definition())
3454         return static_cast<reducibility_hints const &>(
3455             cnstr_get_ref(to_val(), 2));
3456     else
3457         return *g_opaque;
3458 }
3459
3460 bool constant_info::is_unsafe() const {
3461     switch (kind()) {
3462     case constant_info_kind::Axiom:
3463         return to_axiom_val().is_unsafe();
3464     case constant_info_kind::Definition:
3465         return to_definition_val().get_safety() ==
3466             definition_safety::unsafe;
3467     case constant_info_kind::Theorem:
3468         return false;
3469     case constant_info_kind::Opaque:
3470         return to_opaque_val().is_unsafe();
3471     case constant_info_kind::Quot:
3472         return false;
3473     case constant_info_kind::Inductive:
3474         return to_inductive_val().is_unsafe();
3475     case constant_info_kind::Constructor:
3476         return to_constructor_val().is_unsafe();
3477     case constant_info_kind::Recursor:
3478         return to_recursor_val().is_unsafe();
3479     }
3480     lean_unreachable();
3481 }
3482
3483 void initialize_declaration() {
3484     g_opaque = new reducibility_hints(reducibility_hints::mk_opaque());
3485     mark_persistent(g_opaque->raw());
3486     g_dummy = new declaration(mk_axiom(name(), names(), expr()));
3487     mark_persistent(g_dummy->raw());
3488 }
3489
3490 void finalize_declaration() {
3491     delete g_dummy;
3492     delete g_opaque;
3493 }
3494 } // namespace lean
3495 // :::::::::::::::
3496 // environment.cpp
3497 // :::::::::::::::
3498 /*
3499 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.

```



```

3500 Released under Apache 2.0 license as described in the file LICENSE.
3501
3502 Author: Leonardo de Moura
3503 */
3504 #include <lean/sstream.h>
3505 #include <lean/thread.h>
3506
3507 #include <limits>
3508 #include <utility>
3509 #include <vector>
3510
3511 #include "kernel/environment.h"
3512 #include "kernel/kernel_exception.h"
3513 #include "kernel/quot.h"
3514 #include "kernel/type_checker.h"
3515 #include "util/io.h"
3516 #include "util/map_foreach.h"
3517
3518 namespace lean {
3519 extern "C" object *lean_environment_add(object *, object *);
3520 extern "C" object *lean_mk_empty_environment(uint32, object *);
3521 extern "C" object *lean_environment_find(object *, object *);
3522 extern "C" uint32 lean_environment_trust_level(object *);
3523 extern "C" object *lean_environment_mark_quot_init(object *);
3524 extern "C" uint8 lean_environment_quot_init(object *);
3525 extern "C" object *lean_register_extension(object *);
3526 extern "C" object *lean_get_extension(object *, object *);
3527 extern "C" object *lean_set_extension(object *, object *, object *);
3528 extern "C" object *lean_environment_set_main_module(object *, object *);
3529 extern "C" object *lean_environment_main_module(object *);
3530
3531 environment mk_empty_environment(uint32 trust_lvl) {
3532     return get_io_result<environment>(
3533         lean_mk_empty_environment(trust_lvl, io_mk_world()));
3534 }
3535
3536 environment::environment(unsigned trust_lvl)
3537     : object_ref(mk_empty_environment(trust_lvl)) {}
3538
3539 void environment::set_main_module(name const &n) {
3540     m_obj = lean_environment_set_main_module(m_obj, n.to_obj_arg());
3541 }
3542
3543 name environment::get_main_module() const {
3544     return name(lean_environment_main_module(to_obj_arg()));
3545 }
3546
3547 unsigned environment::trust_lvl() const {
3548     return lean_environment_trust_level(to_obj_arg());
3549 }
3550
3551 bool environment::is_quot_initialized() const {
3552     return lean_environment_quot_init(to_obj_arg()) != 0;
3553 }
3554
3555 void environment::mark_quot_initialized() {
3556     m_obj = lean_environment_mark_quot_init(m_obj);
3557 }
3558
3559 optional<constant_info> environment::find(name const &n) const {
3560     return to_optional<constant_info>(
3561         lean_environment_find(to_obj_arg(), n.to_obj_arg()));
3562 }
3563
3564 constant_info environment::get(name const &n) const {
3565     object *o = lean_environment_find(to_obj_arg(), n.to_obj_arg());
3566     if (is_scalar(o)) throw unknown_constant_exception(*this, n);
3567     constant_info r(cnstr_get(o, 0), true);
3568     dec(o);
3569     return r;

```

```

3570 }
3571
3572 static void check_no_metavar(environment const &env, name const &n,
3573                             expr const &e) {
3574     if (has_metavar(e)) throw declaration_has_metavars_exception(env, n, e);
3575 }
3576
3577 static void check_no_fvar(environment const &env, name const &n,
3578                             expr const &e) {
3579     if (has_fvar(e)) throw declaration_has_free_vars_exception(env, n, e);
3580 }
3581
3582 void check_no_metavar_no_fvar(environment const &env, name const &n,
3583                             expr const &e) {
3584     check_no_metavar(env, n, e);
3585     check_no_fvar(env, n, e);
3586 }
3587
3588 static void check_name(environment const &env, name const &n) {
3589     if (env.find(n)) throw already_declared_exception(env, n);
3590 }
3591
3592 void environment::check_name(name const &n) const {
3593     ::lean::check_name(*this, n);
3594 }
3595
3596 static void check_duplicated_univ_params(environment const &env, names ls) {
3597     while (!is_nil(ls)) {
3598         auto const &p = head(ls);
3599         ls = tail(ls);
3600         if (std::find(ls.begin(), ls.end(), p) != ls.end()) {
3601             throw kernel_exception(
3602                 env, sstream() << "failed to add declaration to environment, "
3603                 << "duplicate universe level parameter: '" << p
3604                 << "'");
3605         }
3606     }
3607 }
3608
3609 void environment::check_duplicated_univ_params(names ls) const {
3610     ::lean::check_duplicated_univ_params(*this, ls);
3611 }
3612
3613 static void check_constant_val(environment const &env, constant_val const &v,
3614                             type_checker &checker) {
3615     check_name(env, v.get_name());
3616     check_duplicated_univ_params(env, v.get_lparams());
3617     check_no_metavar_no_fvar(env, v.get_name(), v.get_type());
3618     expr sort = checker.check(v.get_type(), v.get_lparams());
3619     checker.ensure_sort(sort, v.get_type());
3620 }
3621
3622 static void check_constant_val(environment const &env, constant_val const &v,
3623                             bool safe_only) {
3624     type_checker checker(env, safe_only);
3625     check_constant_val(env, v, checker);
3626 }
3627
3628 void environment::add_core(constant_info const &info) {
3629     m_obj = lean_environment_add(m_obj, info.to_obj_arg());
3630 }
3631
3632 environment environment::add(constant_info const &info) const {
3633     return environment(lean_environment_add(to_obj_arg(), info.to_obj_arg()));
3634 }
3635
3636 environment environment::add_axiom(declaration const &d, bool check) const {
3637     axiom_val const &v = d.to_axiom_val();
3638     if (check) check_constant_val(*this, v.to_constant_val(), !d.is_unsafe());
3639     return add(constant_info(d));

```

```

3640 }
3641
3642 environment environment::add_definition(declaration const &d,
3643                                     bool check) const {
3644     definition_val const &v = d.to_definition_val();
3645     if (v.is_unsafe()) {
3646         /* Meta definition can be recursive.
3647            So, we check the header, add, and then type check the body. */
3648         if (check) {
3649             bool safe_only = false;
3650             type_checker checker(*this, safe_only);
3651             check_constant_val(*this, v.to_constant_val(), checker);
3652         }
3653         environment new_env = add(constant_info(d));
3654         if (check) {
3655             bool safe_only = false;
3656             type_checker checker(new_env, safe_only);
3657             check_no_metavar_no_fvar(new_env, v.get_name(), v.get_value());
3658             expr val_type = checker.check(v.get_value(), v.get_lparams());
3659             if (!checker.is_def_eq(val_type, v.get_type()))
3660                 throw definition_type_mismatch_exception(new_env, d, val_type);
3661         }
3662         return new_env;
3663     } else {
3664         if (check) {
3665             type_checker checker(*this);
3666             check_constant_val(*this, v.to_constant_val(), checker);
3667             check_no_metavar_no_fvar(*this, v.get_name(), v.get_value());
3668             expr val_type = checker.check(v.get_value(), v.get_lparams());
3669             if (!checker.is_def_eq(val_type, v.get_type()))
3670                 throw definition_type_mismatch_exception(*this, d, val_type);
3671         }
3672         return add(constant_info(d));
3673     }
3674 }
3675
3676 environment environment::add_theorem(declaration const &d, bool check) const {
3677     theorem_val const &v = d.to_theorem_val();
3678     if (check) {
3679         // TODO(Leo): we must add support for handling tasks here
3680         type_checker checker(*this);
3681         check_constant_val(*this, v.to_constant_val(), checker);
3682         check_no_metavar_no_fvar(*this, v.get_name(), v.get_value());
3683         expr val_type = checker.check(v.get_value(), v.get_lparams());
3684         if (!checker.is_def_eq(val_type, v.get_type()))
3685             throw definition_type_mismatch_exception(*this, d, val_type);
3686     }
3687     return add(constant_info(d));
3688 }
3689
3690 environment environment::add_opaque(declaration const &d, bool check) const {
3691     opaque_val const &v = d.to_opaque_val();
3692     if (check) {
3693         type_checker checker(*this);
3694         check_constant_val(*this, v.to_constant_val(), checker);
3695         expr val_type = checker.check(v.get_value(), v.get_lparams());
3696         if (!checker.is_def_eq(val_type, v.get_type()))
3697             throw definition_type_mismatch_exception(*this, d, val_type);
3698     }
3699     return add(constant_info(d));
3700 }
3701
3702 environment environment::add_mutual(declaration const &d, bool check) const {
3703     definition_vals const &vs = d.to_definition_vals();
3704     if (empty(vs))
3705         throw kernel_exception(*this, "invalid empty mutual definition");
3706     definition_safety safety = head(vs).get_safety();
3707     if (safety == definition_safety::safe)
3708         throw kernel_exception(*this,
3709                               "invalid mutual definition, declaration is not "

```

```

3710                                     "tagged as unsafe/partial");
3711     bool safe_only = safety == definition_safety::partial;
3712     /* Check declarations header */
3713     if (check) {
3714         type_checker checker(*this, safe_only);
3715         for (definition_val const &v : vs) {
3716             if (v.get_safety() != safety)
3717                 throw kernel_exception(
3718                     *this,
3719                     "invalid mutual definition, declarations must have the "
3720                     "same safety annotation");
3721             check_constant_val(*this, v.to_constant_val(), checker);
3722         }
3723     }
3724     /* Add declarations */
3725     environment new_env = *this;
3726     for (definition_val const &v : vs) {
3727         new_env.add_core(constant_info(v));
3728     }
3729     /* Check actual definitions */
3730     if (check) {
3731         type_checker checker(new_env, safe_only);
3732         for (definition_val const &v : vs) {
3733             check_no_metavar_no_fvar(new_env, v.get_name(), v.get_value());
3734             expr val_type = checker.check(v.get_value(), v.get_lparams());
3735             if (!checker.is_def_eq(val_type, v.get_type()))
3736                 throw definition_type_mismatch_exception(new_env, d, val_type);
3737         }
3738     }
3739     return new_env;
3740 }
3741
3742 environment environment::add(declaration const &d, bool check) const {
3743     switch (d.kind()) {
3744         case declaration_kind::Axiom:
3745             return add_axiom(d, check);
3746         case declaration_kind::Definition:
3747             return add_definition(d, check);
3748         case declaration_kind::Theorem:
3749             return add_theorem(d, check);
3750         case declaration_kind::Opaque:
3751             return add_opaque(d, check);
3752         case declaration_kind::MutualDefinition:
3753             return add_mutual(d, check);
3754         case declaration_kind::Quot:
3755             return add_quot();
3756         case declaration_kind::Inductive:
3757             return add_inductive(d);
3758     }
3759     lean_unreachable();
3760 }
3761
3762 extern "C" object *lean_add_decl(object *env, object *decl) {
3763     return catch_kernel_exceptions<environment>(
3764         [&]() { return environment(env).add(declaration(decl, true)); });
3765 }
3766
3767 void environment::for_each_constant(
3768     std::function<void(constant_info const &d)> const &f) const {
3769     smap_foreach(cnstr_get(raw(), 1), [&](object *, object *v) {
3770         constant_info cinfo(v, true);
3771         f(cinfo);
3772     });
3773 }
3774
3775 extern "C" obj_res lean_display_stats(obj_arg env, obj_arg w);
3776
3777 void environment::display_stats() const {
3778     dec_ref(lean_display_stats(to_obj_arg(), io_mk_world()));
3779 }

```

```

3780
3781 void initialize_environment() {}
3782
3783 void finalize_environment() {}
3784 } // namespace lean
3785 // ::::::::::::::
3786 // equiv_manager.cpp
3787 // ::::::::::::::
3788 /*
3789 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
3790 Released under Apache 2.0 license as described in the file LICENSE.
3791
3792 Author: Leonardo de Moura
3793 */
3794 #include <lean/flet.h>
3795 #include <lean/interrupt.h>
3796
3797 #include "kernel/equiv_manager.h"
3798
3799 namespace lean {
3800 auto equiv_manager::mk_node() -> node_ref {
3801     node_ref r = m_nodes.size();
3802     node n;
3803     n.m_parent = r;
3804     n.m_rank = 0;
3805     m_nodes.push_back(n);
3806     return r;
3807 }
3808
3809 auto equiv_manager::find(node_ref n) -> node_ref {
3810     while (true) {
3811         node_ref p = m_nodes[n].m_parent;
3812         if (p == n) return p;
3813         n = p;
3814     }
3815 }
3816
3817 void equiv_manager::merge(node_ref n1, node_ref n2) {
3818     node_ref r1 = find(n1);
3819     node_ref r2 = find(n2);
3820     if (r1 != r2) {
3821         node &ref1 = m_nodes[r1];
3822         node &ref2 = m_nodes[r2];
3823         if (ref1.m_rank < ref2.m_rank) {
3824             ref1.m_parent = r2;
3825         } else if (ref1.m_rank > ref2.m_rank) {
3826             ref2.m_parent = r1;
3827         } else {
3828             ref2.m_parent = r1;
3829             ref1.m_rank++;
3830         }
3831     }
3832 }
3833
3834 auto equiv_manager::to_node(expr const &e) -> node_ref {
3835     auto it = m_to_node.find(e);
3836     if (it != m_to_node.end()) return it->second;
3837     node_ref r = mk_node();
3838     m_to_node.insert(mk_pair(e, r));
3839     return r;
3840 }
3841
3842 bool equiv_manager::is_equiv_core(expr const &a, expr const &b) {
3843     if (is_eqp(a, b)) return true;
3844     if (m_use_hash && hash(a) != hash(b)) return false;
3845     if (is_bvar(a) && is_bvar(b)) return bvar_idx(a) == bvar_idx(b);
3846     node_ref r1 = find(to_node(a));
3847     node_ref r2 = find(to_node(b));
3848     if (r1 == r2) return true;
3849     // fall back to structural equality

```

```

3850     if (a.kind() != b.kind()) return false;
3851     check_system("expression equivalence test");
3852     bool result = false;
3853     switch (a.kind()) {
3854     case expr_kind::BVar:
3855         lean_unreachable(); // LCOV_EXCL_LINE
3856     case expr_kind::Const:
3857         result = const_name(a) == const_name(b) &&
3858                 compare(const_levels(a), const_levels(b),
3859                        [](level const &l1, level const &l2) {
3860                            return l1 == l2;
3861                        });
3862         break;
3863     case expr_kind::MVar:
3864         result = mvar_name(a) == mvar_name(b);
3865         break;
3866     case expr_kind::FVar:
3867         result = fvar_name(a) == fvar_name(b);
3868         break;
3869     case expr_kind::App:
3870         result = is_equiv_core(app_fn(a), app_fn(b)) &&
3871                 is_equiv_core(app_arg(a), app_arg(b));
3872         break;
3873     case expr_kind::Lambda:
3874     case expr_kind::Pi:
3875         result = is_equiv_core(binding_domain(a), binding_domain(b)) &&
3876                 is_equiv_core(binding_body(a), binding_body(b));
3877         break;
3878     case expr_kind::Sort:
3879         result = sort_level(a) == sort_level(b);
3880         break;
3881     case expr_kind::Lit:
3882         result = lit_value(a) == lit_value(b);
3883         break;
3884     case expr_kind::MData:
3885         result = is_equiv_core(mdata_expr(a), mdata_expr(b));
3886         break;
3887     case expr_kind::Proj:
3888         result = is_equiv_core(proj_expr(a), proj_expr(b)) &&
3889                 proj_idx(a) == proj_idx(b);
3890         break;
3891     case expr_kind::Let:
3892         result = is_equiv_core(let_type(a), let_type(b)) &&
3893                 is_equiv_core(let_value(a), let_value(b)) &&
3894                 is_equiv_core(let_body(a), let_body(b));
3895         break;
3896     }
3897     if (result) merge(r1, r2);
3898     return result;
3899 }
3900
3901 bool equiv_manager::is_equiv(expr const &a, expr const &b, bool use_hash) {
3902     flet<bool> set(m_use_hash, use_hash);
3903     return is_equiv_core(a, b);
3904 }
3905
3906 void equiv_manager::add_equiv(expr const &e1, expr const &e2) {
3907     node_ref r1 = to_node(e1);
3908     node_ref r2 = to_node(e2);
3909     merge(r1, r2);
3910 }
3911 } // namespace lean
3912 // ::::::::::::::
3913 // expr_cache.cpp
3914 // ::::::::::::::
3915 /*
3916 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
3917 Released under Apache 2.0 license as described in the file LICENSE.
3918
3919 Author: Leonardo de Moura

```



```

3920 */
3921 #include "kernel/expr_cache.h"
3922
3923 namespace lean {
3924 expr *expr_cache::find(expr const &e) {
3925     unsigned i = hash(e) % m_capacity;
3926     if (m_cache[i].m_expr && is_bi_equal(*m_cache[i].m_expr, e))
3927         return &m_cache[i].m_result;
3928     else
3929         return nullptr;
3930 }
3931
3932 void expr_cache::insert(expr const &e, expr const &v) {
3933     unsigned i = hash(e) % m_capacity;
3934     if (!m_cache[i].m_expr) m_used.push_back(i);
3935     m_cache[i].m_expr = e;
3936     m_cache[i].m_result = v;
3937 }
3938
3939 void expr_cache::clear() {
3940     for (unsigned i : m_used) {
3941         m_cache[i].m_expr = none_expr();
3942         m_cache[i].m_result = expr();
3943     }
3944     m_used.clear();
3945 }
3946 } // namespace lean
3947 // ::::::::::::::
3948 // expr.cpp
3949 // ::::::::::::::
3950 /*
3951 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
3952 Released under Apache 2.0 license as described in the file LICENSE.
3953
3954 Author: Leonardo de Moura
3955         Soonho Kong
3956 */
3957 #include <lean/hash.h>
3958
3959 #include <algorithm>
3960 #include <limits>
3961 #include <sstream>
3962 #include <string>
3963 #include <vector>
3964
3965 #include "kernel/abstract.h"
3966 #include "kernel/expr.h"
3967 #include "kernel/expr_eq_fn.h"
3968 #include "kernel/expr_sets.h"
3969 #include "kernel/for_each_fn.h"
3970 #include "kernel/instantiate.h"
3971 #include "kernel/replace_fn.h"
3972 #include "util/buffer.h"
3973 #include "util/list_fn.h"
3974
3975 namespace lean {
3976 /* Expression literal values */
3977 literal::literal(char const *v)
3978     : object_ref(mk_cnstr(static_cast<unsigned>(literal_kind::String),
3979                          mk_string(v))) {}
3980
3981 literal::literal(unsigned v)
3982     : object_ref(
3983         mk_cnstr(static_cast<unsigned>(literal_kind::Nat), mk_nat_obj(v))) {}
3984
3985 literal::literal(mpz const &v)
3986     : object_ref(
3987         mk_cnstr(static_cast<unsigned>(literal_kind::Nat), mk_nat_obj(v))) {}
3988
3989 literal::literal(nat const &v)

```

```

3990 : object_ref(mk_cnstr(static_cast<unsigned>(literal_kind::Nat), v)) {}
3991
3992 bool operator==(literal const &a, literal const &b) {
3993     if (a.kind() != b.kind()) return false;
3994     switch (a.kind()) {
3995         case literal_kind::String:
3996             return a.get_string() == b.get_string();
3997         case literal_kind::Nat:
3998             return a.get_nat() == b.get_nat();
3999     }
4000     lean_unreachable();
4001 }
4002
4003 bool operator<(literal const &a, literal const &b) {
4004     if (a.kind() != b.kind())
4005         return static_cast<unsigned>(a.kind()) <
4006                static_cast<unsigned>(b.kind());
4007     switch (a.kind()) {
4008         case literal_kind::String:
4009             return a.get_string() < b.get_string();
4010         case literal_kind::Nat:
4011             return a.get_nat() < b.get_nat();
4012     }
4013     lean_unreachable();
4014 }
4015
4016 bool is_atomic(expr const &e) {
4017     switch (e.kind()) {
4018         case expr_kind::Const:
4019         case expr_kind::Sort:
4020         case expr_kind::BVar:
4021         case expr_kind::Lit:
4022         case expr_kind::MVar:
4023         case expr_kind::FVar:
4024             return true;
4025         case expr_kind::App:
4026         case expr_kind::Lambda:
4027         case expr_kind::Pi:
4028         case expr_kind::Let:
4029         case expr_kind::MData:
4030         case expr_kind::Proj:
4031             return false;
4032     }
4033     lean_unreachable(); // LCOV_EXCL_LINE
4034 }
4035
4036 extern "C" uint8 lean_expr_binder_info(object *e);
4037 binder_info binding_info(expr const &e) {
4038     return static_cast<binder_info>(lean_expr_binder_info(e.to_obj_arg()));
4039 }
4040
4041 extern "C" object *lean_lit_type(obj_arg e);
4042 expr lit_type(literal const &lit) {
4043     return expr(lean_lit_type(lit.to_obj_arg()));
4044 }
4045
4046 extern "C" uint8 lean_expr_hash(obj_arg e);
4047 unsigned hash(expr const &e) { return lean_expr_hash(e.to_obj_arg()); }
4048
4049 extern "C" uint8 lean_expr_has_fvar(obj_arg e);
4050 bool has_fvar(expr const &e) { return lean_expr_has_fvar(e.to_obj_arg()); }
4051
4052 extern "C" uint8 lean_expr_has_expr_mvar(obj_arg e);
4053 bool has_expr_mvar(expr const &e) {
4054     return lean_expr_has_expr_mvar(e.to_obj_arg());
4055 }
4056
4057 extern "C" uint8 lean_expr_has_level_mvar(obj_arg e);
4058 bool has_univ_mvar(expr const &e) {
4059     return lean_expr_has_level_mvar(e.to_obj_arg());

```

```

4060 }
4061
4062 extern "C" uint8 lean_expr_has_level_param(obj_arg e);
4063 bool has_univ_param(expr const &e) {
4064     return lean_expr_has_level_param(e.to_obj_arg());
4065 }
4066
4067 extern "C" unsigned lean_expr_loose_bvar_range(object *e);
4068 unsigned get_loose_bvar_range(expr const &e) {
4069     return lean_expr_loose_bvar_range(e.to_obj_arg());
4070 }
4071
4072 // =====
4073 // Constructors
4074
4075 static expr *g_dummy = nullptr;
4076
4077 static expr const &get_dummy() {
4078     if (!g_dummy) {
4079         g_dummy = new expr(mk_constant("__expr_for_default_constructor__"));
4080         mark_persistent(g_dummy->raw());
4081     }
4082     return *g_dummy;
4083 }
4084
4085 expr::expr() : expr(get_dummy()) {}
4086
4087 extern "C" object *lean_expr_mk_lit(obj_arg l);
4088 expr mk_lit(literal const &l) { return expr(lean_expr_mk_lit(l.to_obj_arg())); }
4089
4090 extern "C" object *lean_expr_mk_mdata(obj_arg m, obj_arg e);
4091 expr mk_mdata(kvmap const &m, expr const &e) {
4092     return expr(lean_expr_mk_mdata(m.to_obj_arg(), e.to_obj_arg()));
4093 }
4094
4095 extern "C" object *lean_expr_mk_proj(obj_arg s, obj_arg idx, obj_arg e);
4096 expr mk_proj(name const &s, nat const &idx, expr const &e) {
4097     return expr(
4098         lean_expr_mk_proj(s.to_obj_arg(), idx.to_obj_arg(), e.to_obj_arg()));
4099 }
4100
4101 extern "C" object *lean_expr_mk_bvar(obj_arg idx);
4102 expr mk_bvar(nat const &idx) {
4103     return expr(lean_expr_mk_bvar(idx.to_obj_arg()));
4104 }
4105
4106 extern "C" object *lean_expr_mk_fvar(obj_arg n);
4107 expr mk_fvar(name const &n) { return expr(lean_expr_mk_fvar(n.to_obj_arg())); }
4108
4109 extern "C" object *lean_expr_mk_mvar(object *n);
4110 expr mk_mvar(name const &n) { return expr(lean_expr_mk_mvar(n.to_obj_arg())); }
4111
4112 extern "C" object *lean_expr_mk_const(obj_arg n, obj_arg ls);
4113 expr mk_const(name const &n, levels const &ls) {
4114     return expr(lean_expr_mk_const(n.to_obj_arg(), ls.to_obj_arg()));
4115 }
4116
4117 extern "C" object *lean_expr_mk_app(obj_arg f, obj_arg a);
4118 expr mk_app(expr const &f, expr const &a) {
4119     return expr(lean_expr_mk_app(f.to_obj_arg(), a.to_obj_arg()));
4120 }
4121
4122 extern "C" object *lean_expr_mk_sort(obj_arg l);
4123 expr mk_sort(level const &l) { return expr(lean_expr_mk_sort(l.to_obj_arg())); }
4124
4125 extern "C" object *lean_expr_mk_lambda(obj_arg n, obj_arg t, obj_arg e,
4126                                         uint8 bi);
4127 expr mk_lambda(name const &n, expr const &t, expr const &e, binder_info bi) {
4128     return expr(lean_expr_mk_lambda(n.to_obj_arg(), t.to_obj_arg(),
4129                                     e.to_obj_arg(), static_cast<uint8>(bi)));
4129 }

```

```

4130 }
4131
4132 extern "C" object *lean_expr_mk_forall(obj_arg n, obj_arg t, obj_arg e,
4133                                         uint8 bi);
4134 expr mk_pi(name const &n, expr const &t, expr const &e, binder_info bi) {
4135     return expr(lean_expr_mk_forall(n.to_obj_arg(), t.to_obj_arg(),
4136                                     e.to_obj_arg(), static_cast<uint8>(bi)));
4137 }
4138
4139 static name *g_default_name = nullptr;
4140 expr mk_arrow(expr const &t, expr const &e) {
4141     return mk_pi(*g_default_name, t, e, mk_binder_info());
4142 }
4143
4144 extern "C" object *lean_expr_mk_let(object *n, object *t, object *v, object *b);
4145 expr mk_let(name const &n, expr const &t, expr const &v, expr const &b) {
4146     return expr(lean_expr_mk_let(n.to_obj_arg(), t.to_obj_arg(), v.to_obj_arg(),
4147                                  b.to_obj_arg()));
4148 }
4149
4150 static expr *g_Prop = nullptr;
4151 static expr *g_Type0 = nullptr;
4152 expr mk_Prop() { return *g_Prop; }
4153 expr mk_Type() { return *g_Type0; }
4154
4155 // =====
4156 // Auxiliary constructors and accessors
4157
4158 expr mk_app(expr const &f, unsigned num_args, expr const *args) {
4159     expr r = f;
4160     for (unsigned i = 0; i < num_args; i++) r = mk_app(r, args[i]);
4161     return r;
4162 }
4163
4164 expr mk_app(unsigned num_args, expr const *args) {
4165     lean_assert(num_args >= 2);
4166     return mk_app(mk_app(args[0], args[1]), num_args - 2, args + 2);
4167 }
4168
4169 expr mk_app(expr const &f, list<expr> const &args) {
4170     buffer<expr> _args;
4171     to_buffer(args, _args);
4172     return mk_app(f, _args);
4173 }
4174
4175 expr mk_rev_app(expr const &f, unsigned num_args, expr const *args) {
4176     expr r = f;
4177     unsigned i = num_args;
4178     while (i > 0) {
4179         --i;
4180         r = mk_app(r, args[i]);
4181     }
4182     return r;
4183 }
4184
4185 expr mk_rev_app(unsigned num_args, expr const *args) {
4186     lean_assert(num_args >= 2);
4187     return mk_rev_app(mk_app(args[num_args - 1], args[num_args - 2]),
4188                      num_args - 2, args);
4189 }
4190
4191 expr const &get_app_args(expr const &e, buffer<expr> &args) {
4192     unsigned sz = args.size();
4193     expr const *it = &e;
4194     while (is_app(*it)) {
4195         args.push_back(app_arg(*it));
4196         it = &(app_fn(*it));
4197     }
4198     std::reverse(args.begin() + sz, args.end());
4199     return *it;

```

```

4200 }
4201
4202 expr const &get_app_args_at_most(expr const &e, unsigned num,
4203                                 buffer<expr> &args) {
4204     unsigned sz = args.size();
4205     expr const *it = &e;
4206     unsigned i = 0;
4207     while (is_app(*it)) {
4208         if (i == num) break;
4209         args.push_back(app_arg(*it));
4210         it = &(app_fn(*it));
4211         i++;
4212     }
4213     std::reverse(args.begin() + sz, args.end());
4214     return *it;
4215 }
4216
4217 expr const &get_app_rev_args(expr const &e, buffer<expr> &args) {
4218     expr const *it = &e;
4219     while (is_app(*it)) {
4220         args.push_back(app_arg(*it));
4221         it = &(app_fn(*it));
4222     }
4223     return *it;
4224 }
4225
4226 expr const &get_app_fn(expr const &e) {
4227     expr const *it = &e;
4228     while (is_app(*it)) {
4229         it = &(app_fn(*it));
4230     }
4231     return *it;
4232 }
4233
4234 unsigned get_app_num_args(expr const &e) {
4235     expr const *it = &e;
4236     unsigned n = 0;
4237     while (is_app(*it)) {
4238         it = &(app_fn(*it));
4239         n++;
4240     }
4241     return n;
4242 }
4243
4244 bool is_arrow(expr const &t) {
4245     if (!is_pi(t)) return false;
4246     if (has_loose_bvars(t)) {
4247         return !has_loose_bvar(binding_body(t), 0);
4248     } else {
4249         lean_assert(has_loose_bvars(binding_body(t)) ==
4250                   has_loose_bvar(binding_body(t), 0));
4251         return !has_loose_bvars(binding_body(t));
4252     }
4253 }
4254
4255 bool is_default_var_name(name const &n) { return n == *g_default_name; }
4256
4257 // =====
4258 // Update
4259
4260 expr update_mdata(expr const &e, expr const &t) {
4261     if (!is_eqp(mdata_expr(e), t))
4262         return mk_mdata(mdata_data(e), t);
4263     else
4264         return e;
4265 }
4266
4267 expr update_proj(expr const &e, expr const &t) {
4268     if (!is_eqp(proj_expr(e), t))
4269         return mk_proj(proj_sname(e), proj_idx(e), t);

```

```

4270     else
4271         return e;
4272 }
4273
4274 expr update_app(expr const &e, expr const &new_fn, expr const &new_arg) {
4275     if (!is_eqp(app_fn(e), new_fn) || !is_eqp(app_arg(e), new_arg))
4276         return mk_app(new_fn, new_arg);
4277     else
4278         return e;
4279 }
4280
4281 expr update_binding(expr const &e, expr const &new_domain,
4282                   expr const &new_body) {
4283     if (!is_eqp(binding_domain(e), new_domain) ||
4284         !is_eqp(binding_body(e), new_body))
4285         return mk_binding(e.kind(), binding_name(e), new_domain, new_body,
4286                           binding_info(e));
4287     else
4288         return e;
4289 }
4290
4291 expr update_binding(expr const &e, expr const &new_domain, expr const &new_body,
4292                   binder_info bi) {
4293     if (!is_eqp(binding_domain(e), new_domain) ||
4294         !is_eqp(binding_body(e), new_body) || bi != binding_info(e))
4295         return mk_binding(e.kind(), binding_name(e), new_domain, new_body, bi);
4296     else
4297         return e;
4298 }
4299
4300 expr update_sort(expr const &e, level const &new_level) {
4301     if (!is_eqp(sort_level(e), new_level))
4302         return mk_sort(new_level);
4303     else
4304         return e;
4305 }
4306
4307 expr update_const(expr const &e, levels const &new_levels) {
4308     if (!is_eqp(const_levels(e), new_levels))
4309         return mk_const(const_name(e), new_levels);
4310     else
4311         return e;
4312 }
4313
4314 expr update_let(expr const &e, expr const &new_type, expr const &new_value,
4315               expr const &new_body) {
4316     if (!is_eqp(let_type(e), new_type) || !is_eqp(let_value(e), new_value) ||
4317         !is_eqp(let_body(e), new_body))
4318         return mk_let(let_name(e), new_type, new_value, new_body);
4319     else
4320         return e;
4321 }
4322
4323 extern "C" object *lean_expr_update_mdata(obj_arg e, obj_arg new_expr) {
4324     if (mdata_expr(TO_REF(expr, e)).raw() != new_expr) {
4325         object *r = lean_expr_mk_mdata(mdata_data(TO_REF(expr, e)).to_obj_arg(),
4326                                       new_expr);
4327         lean_dec_ref(e);
4328         return r;
4329     } else {
4330         lean_dec_ref(new_expr);
4331         return e;
4332     }
4333 }
4334
4335 extern "C" object *lean_expr_update_const(obj_arg e, obj_arg new_levels) {
4336     if (const_levels(TO_REF(expr, e)).raw() != new_levels) {
4337         object *r = lean_expr_mk_const(const_name(TO_REF(expr, e)).to_obj_arg(),
4338                                       new_levels);
4339         lean_dec_ref(e);

```



```

4340     return r;
4341 } else {
4342     lean_dec(new_levels);
4343     return e;
4344 }
4345 }
4346
4347 extern "C" object *lean_expr_update_sort(obj_arg e, obj_arg new_level) {
4348     if (sort_level(T0_REF(expr, e)).raw() != new_level) {
4349         object *r = lean_expr_mk_sort(new_level);
4350         lean_dec_ref(e);
4351         return r;
4352     } else {
4353         lean_dec(new_level);
4354         return e;
4355     }
4356 }
4357
4358 extern "C" object *lean_expr_update_proj(obj_arg e, obj_arg new_expr) {
4359     if (proj_expr(T0_REF(expr, e)).raw() != new_expr) {
4360         object *r =
4361             lean_expr_mk_proj(proj_sname(T0_REF(expr, e)).to_obj_arg(),
4362                             proj_idx(T0_REF(expr, e)).to_obj_arg(), new_expr);
4363         lean_dec_ref(e);
4364         return r;
4365     } else {
4366         lean_dec_ref(new_expr);
4367         return e;
4368     }
4369 }
4370
4371 extern "C" object *lean_expr_update_app(obj_arg e, obj_arg new_fn,
4372                                         obj_arg new_arg) {
4373     if (app_fn(T0_REF(expr, e)).raw() != new_fn ||
4374         app_arg(T0_REF(expr, e)).raw() != new_arg) {
4375         object *r = lean_expr_mk_app(new_fn, new_arg);
4376         lean_dec_ref(e);
4377         return r;
4378     } else {
4379         lean_dec_ref(new_fn);
4380         lean_dec_ref(new_arg);
4381         return e;
4382     }
4383 }
4384
4385 extern "C" object *lean_expr_update_forall(obj_arg e, uint8 new_bininfo,
4386                                           obj_arg new_domain,
4387                                           obj_arg new_body) {
4388     if (binding_domain(T0_REF(expr, e)).raw() != new_domain ||
4389         binding_body(T0_REF(expr, e)).raw() != new_body ||
4390         binding_info(T0_REF(expr, e)) != static_cast<binder_info>(new_bininfo)) {
4391         object *r =
4392             lean_expr_mk_forall(binding_name(T0_REF(expr, e)).to_obj_arg(),
4393                               new_domain, new_body, new_bininfo);
4394         lean_dec_ref(e);
4395         return r;
4396     } else {
4397         lean_dec_ref(new_domain);
4398         lean_dec_ref(new_body);
4399         return e;
4400     }
4401 }
4402
4403 extern "C" object *lean_expr_update_lambda(obj_arg e, uint8 new_bininfo,
4404                                           obj_arg new_domain,
4405                                           obj_arg new_body) {
4406     if (binding_domain(T0_REF(expr, e)).raw() != new_domain ||
4407         binding_body(T0_REF(expr, e)).raw() != new_body ||
4408         binding_info(T0_REF(expr, e)) != static_cast<binder_info>(new_bininfo)) {
4409         object *r =

```

```

4410         lean_expr_mk_lambda(binding_name(T0_REF(expr, e)).to_obj_arg(),
4411                             new_domain, new_body, new_binfo);
4412     lean_dec_ref(e);
4413     return r;
4414 } else {
4415     lean_dec_ref(new_domain);
4416     lean_dec_ref(new_body);
4417     return e;
4418 }
4419 }
4420
4421 extern "C" object *lean_expr_update_let(obj_arg e, obj_arg new_type,
4422                                         obj_arg new_val, obj_arg new_body) {
4423     if (let_type(T0_REF(expr, e)).raw() != new_type ||
4424         let_value(T0_REF(expr, e)).raw() != new_val ||
4425         let_body(T0_REF(expr, e)).raw() != new_body) {
4426         object *r = lean_expr_mk_let(let_name(T0_REF(expr, e)).to_obj_arg(),
4427                                     new_type, new_val, new_body);
4428         lean_dec_ref(e);
4429         return r;
4430     } else {
4431         lean_dec_ref(new_type);
4432         lean_dec_ref(new_val);
4433         lean_dec_ref(new_body);
4434         return e;
4435     }
4436 }
4437
4438 // =====
4439 // Loose bound variable management
4440
4441 static bool has_loose_bvars_in_domain(expr const &b, unsigned vidx,
4442                                       bool strict) {
4443     if (is_pi(b)) {
4444         if (has_loose_bvar(binding_domain(b), vidx)) {
4445             if (is_explicit(binding_info(b))) {
4446                 return true;
4447             } else if (has_loose_bvars_in_domain(binding_body(b), 0, strict)) {
4448                 // "Transitivity": vidx occurs in current implicit argument, so
4449                 // we search for current argument in the body.
4450                 return true;
4451             }
4452         }
4453         // finally we search for vidx in the body
4454         return has_loose_bvars_in_domain(binding_body(b), vidx + 1, strict);
4455     } else if (!strict) {
4456         return has_loose_bvar(b, vidx);
4457     } else {
4458         return false;
4459     }
4460 }
4461
4462 bool has_loose_bvar(expr const &e, unsigned i) {
4463     if (!has_loose_bvars(e)) return false;
4464     bool found = false;
4465     for_each(e, [&](expr const &e, unsigned offset) {
4466         if (found) return false; // already found
4467         unsigned n_i = i + offset;
4468         if (n_i < i) return false; // overflow, vidx can't be >= max unsigned
4469         if (n_i >= get_loose_bvar_range(e))
4470             return false; // expression e does not contain bound variables with
4471                           // idx >= n_i
4472         if (is_var(e)) {
4473             nat const &vidx = bvar_idx(e);
4474             if (vidx == n_i) found = true;
4475         }
4476         return true; // continue search
4477     });
4478     return found;
4479 }

```

```

4480
4481 extern "C" uint8 lean_expr_has_loose_bvar(b_obj_arg e, b_obj_arg i) {
4482     if (!lean_is_scalar(i)) return false;
4483     return has_loose_bvar(TO_REF(expr, e), lean_unbox(i));
4484 }
4485
4486 expr lower_loose_bvars(expr const &e, unsigned s, unsigned d) {
4487     if (d == 0 || s >= get_loose_bvar_range(e)) return e;
4488     lean_assert(s >= d);
4489     return replace(e, [=](expr const &e, unsigned offset) -> optional<expr> {
4490         unsigned s1 = s + offset;
4491         if (s1 < s)
4492             return some_expr(e); // overflow, vidx can't be >= max unsigned
4493         if (s1 >= get_loose_bvar_range(e))
4494             return some_expr(e); // expression e does not contain bound
4495                                     // variables with idx >= s1
4496         if (is_bvar(e) && bvar_idx(e) >= s1) {
4497             lean_assert(bvar_idx(e) >= offset + d);
4498             return some_expr(mk_bvar(bvar_idx(e) - nat(d)));
4499         } else {
4500             return none_expr();
4501         }
4502     });
4503 }
4504
4505 expr lower_loose_bvars(expr const &e, unsigned d) {
4506     return lower_loose_bvars(e, d, d);
4507 }
4508
4509 extern "C" object *lean_expr_lower_loose_bvars(b_obj_arg e, b_obj_arg s,
4510                                                b_obj_arg d) {
4511     if (!lean_is_scalar(s) || !lean_is_scalar(d) ||
4512         lean_unbox(s) < lean_unbox(d)) {
4513         lean_inc(e);
4514         return e;
4515     }
4516     return lower_loose_bvars(TO_REF(expr, e), lean_unbox(s), lean_unbox(d))
4517         .steal();
4518 }
4519
4520 expr lift_loose_bvars(expr const &e, unsigned s, unsigned d) {
4521     if (d == 0 || s >= get_loose_bvar_range(e)) return e;
4522     return replace(e, [=](expr const &e, unsigned offset) -> optional<expr> {
4523         unsigned s1 = s + offset;
4524         if (s1 < s)
4525             return some_expr(e); // overflow, vidx can't be >= max unsigned
4526         if (s1 >= get_loose_bvar_range(e))
4527             return some_expr(e); // expression e does not contain bound
4528                                     // variables with idx >= s1
4529         if (is_var(e) && bvar_idx(e) >= s + offset) {
4530             return some_expr(mk_bvar(bvar_idx(e) + nat(d)));
4531         } else {
4532             return none_expr();
4533         }
4534     });
4535 }
4536
4537 expr lift_loose_bvars(expr const &e, unsigned d) {
4538     return lift_loose_bvars(e, 0, d);
4539 }
4540
4541 extern "C" object *lean_expr_lift_loose_bvars(b_obj_arg e, b_obj_arg s,
4542                                                b_obj_arg d) {
4543     if (!lean_is_scalar(s) || !lean_is_scalar(d)) {
4544         lean_inc(e);
4545         return e;
4546     }
4547     return lift_loose_bvars(TO_REF(expr, e), lean_unbox(s), lean_unbox(d))
4548         .steal();
4549 }

```

```

4550
4551 // =====
4552 // Implicit argument inference
4553
4554 expr infer_implicit(expr const &t, unsigned num_params, bool strict) {
4555     if (num_params == 0) {
4556         return t;
4557     } else if (is_pi(t)) {
4558         expr new_body = infer_implicit(binding_body(t), num_params - 1, strict);
4559         if (!is_explicit(binding_info(t))) {
4560             // argument is already marked as implicit
4561             return update_binding(t, binding_domain(t), new_body);
4562         } else if (has_loose_bvars_in_domain(new_body, 0, strict)) {
4563             return update_binding(t, binding_domain(t), new_body,
4564                                   mk_implicit_binder_info());
4565         } else {
4566             return update_binding(t, binding_domain(t), new_body);
4567         }
4568     } else {
4569         return t;
4570     }
4571 }
4572
4573 expr infer_implicit(expr const &t, bool strict) {
4574     return infer_implicit(t, std::numeric_limits<unsigned>::max(), strict);
4575 }
4576
4577 // =====
4578 // Initialization & Finalization
4579
4580 void initialize_expr() {
4581     get_dummy();
4582     g_default_name = new name("a");
4583     mark_persistent(g_default_name->raw());
4584     g_Type0 = new expr(mk_sort(mk_level_one()));
4585     mark_persistent(g_Type0->raw());
4586     g_Prop = new expr(mk_sort(mk_level_zero()));
4587     mark_persistent(g_Prop->raw());
4588     /* TODO(Leo): add support for builtin constants in the kernel.
4589        Something similar to what we have in the library directory. */
4590 }
4591
4592 void finalize_expr() {
4593     delete g_Prop;
4594     delete g_Type0;
4595     delete g_dummy;
4596     delete g_default_name;
4597 }
4598
4599 // =====
4600 // Legacy
4601
4602 optional<expr> has_expr_metavar_strict(expr const &e) {
4603     if (!has_expr_metavar(e)) return none_expr();
4604     optional<expr> r;
4605     for_each(e, [&](expr const &e, unsigned) {
4606         if (r || !has_expr_metavar(e)) return false;
4607         if (is_metavar_app(e)) {
4608             r = e;
4609             return false;
4610         }
4611     });
4612     return r;
4613 }
4614 } // namespace lean
4615 // ::::::::::::::
4616 // expr_eq_fn.cpp
4617 // ::::::::::::::
4618 /*

```

```

4620 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
4621 Released under Apache 2.0 license as described in the file LICENSE.
4622
4623 Author: Leonardo de Moura
4624 */
4625 #include <lean/interrupt.h>
4626 #include <lean/thread.h>
4627
4628 #include <memory>
4629 #include <vector>
4630
4631 #include "kernel/expr.h"
4632 #include "kernel/expr_sets.h"
4633
4634 #ifndef LEAN_EQ_CACHE_CAPACITY
4635 #define LEAN_EQ_CACHE_CAPACITY 1024 * 8
4636 #endif
4637
4638 namespace lean {
4639 struct eq_cache {
4640     struct entry {
4641         object *m_a;
4642         object *m_b;
4643         entry() : m_a(nullptr), m_b(nullptr) {}
4644     };
4645     unsigned m_capacity;
4646     std::vector<entry> m_cache;
4647     std::vector<unsigned> m_used;
4648     eq_cache()
4649         : m_capacity(LEAN_EQ_CACHE_CAPACITY), m_cache(LEAN_EQ_CACHE_CAPACITY) {}
4650
4651     bool check(expr const &a, expr const &b) {
4652         if (!is_shared(a) || !is_shared(b)) return false;
4653         unsigned i = hash(hash(a), hash(b)) % m_capacity;
4654         if (m_cache[i].m_a == a.raw() && m_cache[i].m_b == b.raw()) {
4655             return true;
4656         } else {
4657             if (m_cache[i].m_a == nullptr) m_used.push_back(i);
4658             m_cache[i].m_a = a.raw();
4659             m_cache[i].m_b = b.raw();
4660             return false;
4661         }
4662     }
4663
4664     void clear() {
4665         for (unsigned i : m_used) m_cache[i].m_a = nullptr;
4666         m_used.clear();
4667     }
4668 };
4669
4670 /* CACHE_RESET: No */
4671 MK_THREAD_LOCAL_GET_DEF(eq_cache, get_eq_cache);
4672
4673 /** \brief Functional object for comparing expressions.
4674
4675     Remark if CompareBinderInfo is true, then functional object will also
4676     compare binder information attached to lambda and Pi expressions */
4677 template <bool CompareBinderInfo>
4678 class expr_eq_fn {
4679     eq_cache &m_cache;
4680
4681     static void check_system() {
4682         ::lean::check_system("expression equality test");
4683     }
4684
4685     bool apply(expr const &a, expr const &b) {
4686         if (is_eqp(a, b)) return true;
4687         if (hash(a) != hash(b)) return false;
4688         if (a.kind() != b.kind()) return false;
4689         if (is_bvar(a)) return bvar_idx(a) == bvar_idx(b);

```

```

4690     if (m_cache.check(a, b)) return true;
4691     switch (a.kind()) {
4692     case expr_kind::BVar:
4693         lean_unreachable(); // LCOV_EXCL_LINE
4694     case expr_kind::MData:
4695         return apply(mdata_expr(a), mdata_expr(b)) &&
4696                mdata_data(a) == mdata_data(b);
4697     case expr_kind::Proj:
4698         return apply(proj_expr(a), proj_expr(b)) &&
4699                proj_sname(a) == proj_sname(b) &&
4700                proj_idx(a) == proj_idx(b);
4701     case expr_kind::Lit:
4702         return lit_value(a) == lit_value(b);
4703     case expr_kind::Const:
4704         return const_name(a) == const_name(b) &&
4705                compare(const_levels(a), const_levels(b),
4706                       [](level const &l1, level const &l2) {
4707                           return l1 == l2;
4708                       });
4709     case expr_kind::MVar:
4710         return mvar_name(a) == mvar_name(b);
4711     case expr_kind::FVar:
4712         return fvar_name(a) == fvar_name(b);
4713     case expr_kind::App:
4714         check_system();
4715         return apply(app_fn(a), app_fn(b)) &&
4716                apply(app_arg(a), app_arg(b));
4717     case expr_kind::Lambda:
4718     case expr_kind::Pi:
4719         check_system();
4720         return apply(binding_domain(a), binding_domain(b)) &&
4721                apply(binding_body(a), binding_body(b)) &&
4722                (!CompareBinderInfo ||
4723                 binding_name(a) == binding_name(b)) &&
4724                (!CompareBinderInfo ||
4725                 binding_info(a) == binding_info(b));
4726     case expr_kind::Let:
4727         check_system();
4728         return apply(let_type(a), let_type(b)) &&
4729                apply(let_value(a), let_value(b)) &&
4730                apply(let_body(a), let_body(b)) &&
4731                (!CompareBinderInfo || let_name(a) == let_name(b));
4732     case expr_kind::Sort:
4733         return sort_level(a) == sort_level(b);
4734     }
4735     lean_unreachable(); // LCOV_EXCL_LINE
4736 }
4737
4738 public:
4739     expr_eq_fn() : m_cache(get_eq_cache()) {}
4740     ~expr_eq_fn() { m_cache.clear(); }
4741     bool operator()(expr const &a, expr const &b) { return apply(a, b); }
4742 };
4743
4744 bool is_equal(expr const &a, expr const &b) {
4745     return expr_eq_fn<false>()(a, b);
4746 }
4747 bool is_bi_equal(expr const &a, expr const &b) {
4748     return expr_eq_fn<true>()(a, b);
4749 }
4750
4751 extern "C" uint8 lean_expr_eqv(b_obj_arg a, b_obj_arg b) {
4752     return expr_eq_fn<false>()(TO_REF(expr, a), TO_REF(expr, b));
4753 }
4754
4755 extern "C" uint8 lean_expr_equal(b_obj_arg a, b_obj_arg b) {
4756     return expr_eq_fn<true>()(TO_REF(expr, a), TO_REF(expr, b));
4757 }
4758 } // namespace lean
4759 // ::::::::::::::

```



```

4760 // for_each_fn.cpp
4761 // ::::::::::::::
4762 /*
4763 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
4764 Released under Apache 2.0 license as described in the file LICENSE.
4765
4766 Author: Leonardo de Moura
4767 */
4768 #include <lean/flet.h>
4769 #include <lean/interrupt.h>
4770 #include <lean/memory.h>
4771
4772 #include <utility>
4773 #include <vector>
4774
4775 #include "kernel/cache_stack.h"
4776 #include "kernel/for_each_fn.h"
4777
4778 #ifndef LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY
4779 #define LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY 1024 * 8
4780 #endif
4781
4782 namespace lean {
4783 struct for_each_cache {
4784     struct entry {
4785         object const *m_cell;
4786         unsigned m_offset;
4787         entry() : m_cell(nullptr) {}
4788     };
4789     unsigned m_capacity;
4790     std::vector<entry> m_cache;
4791     std::vector<unsigned> m_used;
4792     for_each_cache(unsigned c) : m_capacity(c), m_cache(c) {}
4793
4794     bool visited(expr const &e, unsigned offset) {
4795         unsigned i = hash(hash(e), offset) % m_capacity;
4796         if (m_cache[i].m_cell == e.raw() && m_cache[i].m_offset == offset) {
4797             return true;
4798         } else {
4799             if (m_cache[i].m_cell == nullptr) m_used.push_back(i);
4800             m_cache[i].m_cell = e.raw();
4801             m_cache[i].m_offset = offset;
4802             return false;
4803         }
4804     }
4805
4806     void clear() {
4807         for (unsigned i : m_used) m_cache[i].m_cell = nullptr;
4808         m_used.clear();
4809     }
4810 };
4811
4812 /* CACHE_RESET: NO */
4813 MK_CACHE_STACK(for_each_cache, LEAN_DEFAULT_FOR_EACH_CACHE_CAPACITY)
4814
4815 class for_each_fn {
4816     for_each_cache_ref m_cache;
4817     std::function<bool(expr const &, unsigned)> m_f; // NOLINT
4818
4819     void apply(expr const &e, unsigned offset) {
4820         buffer<pair<expr const &, unsigned>> todo;
4821         todo.emplace_back(e, offset);
4822         while (true) {
4823             begin_loop:
4824                 if (todo.empty()) break;
4825                 check_interrupted();
4826                 check_memory("expression traversal");
4827                 auto p = todo.back();
4828                 todo.pop_back();
4829                 expr const &e = p.first;

```

```

4830         unsigned offset = p.second;
4831
4832         switch (e.kind()) {
4833             case expr_kind::Const:
4834             case expr_kind::BVar:
4835             case expr_kind::Sort:
4836                 m_f(e, offset);
4837                 goto begin_loop;
4838             default:
4839                 break;
4840         }
4841
4842         if (is_shared(e) && m_cache->visited(e, offset)) goto begin_loop;
4843
4844         if (!m_f(e, offset)) goto begin_loop;
4845
4846         switch (e.kind()) {
4847             case expr_kind::Const:
4848             case expr_kind::BVar:
4849             case expr_kind::Sort:
4850             case expr_kind::Lit:
4851             case expr_kind::MVar:
4852             case expr_kind::FVar:
4853                 goto begin_loop;
4854             case expr_kind::MData:
4855                 todo.emplace_back(mdata_expr(e), offset);
4856                 goto begin_loop;
4857             case expr_kind::Proj:
4858                 todo.emplace_back(proj_expr(e), offset);
4859                 goto begin_loop;
4860             case expr_kind::App:
4861                 todo.emplace_back(app_arg(e), offset);
4862                 todo.emplace_back(app_fn(e), offset);
4863                 goto begin_loop;
4864             case expr_kind::Lambda:
4865             case expr_kind::Pi:
4866                 todo.emplace_back(binding_body(e), offset + 1);
4867                 todo.emplace_back(binding_domain(e), offset);
4868                 goto begin_loop;
4869             case expr_kind::Let:
4870                 todo.emplace_back(let_body(e), offset + 1);
4871                 todo.emplace_back(let_value(e), offset);
4872                 todo.emplace_back(let_type(e), offset);
4873                 goto begin_loop;
4874         }
4875     }
4876 }
4877
4878 public:
4879     for_each_fn(std::function<bool(expr const &, unsigned)> &&f)
4880         : m_f(f) {} // NOLINT
4881     for_each_fn(std::function<bool(expr const &, unsigned)> const &f)
4882         : m_f(f) {} // NOLINT
4883     void operator()(expr const &e) { apply(e, 0); }
4884 };
4885
4886 void for_each(expr const &e,
4887              std::function<bool(expr const &, unsigned)> &&f) { // NOLINT
4888     return for_each_fn(f)(e);
4889 }
4890 } // namespace lean
4891 // ::::::::::::::
4892 // inductive.cpp
4893 // ::::::::::::::
4894 /*
4895 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
4896 Released under Apache 2.0 license as described in the file LICENSE.
4897
4898 Author: Leonardo de Moura
4899 */

```

```

4900 #include <lean/sstream.h>
4901 #include <lean/utf8.h>
4902
4903 #include "kernel/abstract.h"
4904 #include "kernel/environment.h"
4905 #include "kernel/find_fn.h"
4906 #include "kernel/instantiate.h"
4907 #include "kernel/kernel_exception.h"
4908 #include "kernel/replace_fn.h"
4909 #include "kernel/type_checker.h"
4910 #include "util/name_generator.h"
4911
4912 namespace lean {
4913 static name *g_ind_fresh = nullptr;
4914
4915 /**\ brief Return recursor name for the given inductive datatype name */
4916 name mk_rec_name(name const &I) { return I + name("rec"); }
4917
4918 /** Return the names of all inductive datatypes */
4919 static names get_all_inductive_names(buffer<inductive_type> const &ind_types) {
4920     buffer<name> all_names;
4921     for (inductive_type const &ind_type : ind_types) {
4922         all_names.push_back(ind_type.get_name());
4923     }
4924     return names(all_names);
4925 }
4926
4927 /** Return the names of all inductive datatypes in the given inductive
4928  * declaration */
4929 static names get_all_inductive_names(inductive_decl const &d) {
4930     buffer<inductive_type> ind_types;
4931     to_buffer(d.get_types(), ind_types);
4932     return get_all_inductive_names(ind_types);
4933 }
4934
4935 /** \brief If \c d_name is the name of a non-empty inductive datatype, then
4936     return the name of the first constructor. Return none otherwise. */
4937 static optional<name> get_first_cnstr(environment const &env,
4938                                     name const &d_name) {
4939     constant_info info = env.get(d_name);
4940     if (!info.is_inductive()) return optional<name>();
4941     names const &cnstrs = info.to_inductive_val().get_cnstrs();
4942     if (empty(cnstrs)) return optional<name>();
4943     return optional<name>(head(cnstrs));
4944 }
4945
4946 optional<expr> mk_nullary_cnstr(environment const &env, expr const &type,
4947                               unsigned num_params) {
4948     buffer<expr> args;
4949     expr const &d = get_app_args(type, args);
4950     if (!is_constant(d)) return none_expr();
4951     name const &d_name = const_name(d);
4952     auto cnstr_name = get_first_cnstr(env, d_name);
4953     if (!cnstr_name) return none_expr();
4954     args.shrink(num_params);
4955     return some(mk_app(mk_constant(*cnstr_name, const_levels(d)), args));
4956 }
4957
4958 optional<recursor_rule> get_rec_rule_for(recursor_val const &rec_val,
4959                                         expr const &major) {
4960     expr const &fn = get_app_fn(major);
4961     if (!is_constant(fn)) return optional<recursor_rule>();
4962     for (recursor_rule const &rule : rec_val.get_rules()) {
4963         if (rule.get_cnstr() == const_name(fn))
4964             return optional<recursor_rule>(rule);
4965     }
4966     return optional<recursor_rule>();
4967 }
4968
4969 /** Auxiliary class for adding a mutual inductive datatype declaration. */

```

[illegible]

[illegible]

```

5110         "inductive datatype declaration");
5111
5112     type = tc().ensure_sort(type);
5113
5114     if (first) {
5115         m_result_level = sort_level(type);
5116         m_is_not_zero = is_not_zero(m_result_level);
5117     } else if (!is_equivalent(sort_level(type), m_result_level)) {
5118         throw kernel_exception(
5119             m_env,
5120             "mutually inductive types must live in the same universe");
5121     }
5122
5123     m_ind_cnsts.push_back(mk_constant(ind_type.get_name(), m_levels));
5124     first = false;
5125 }
5126
5127 lean_assert(length(m_levels) == length(m_lparams));
5128 lean_assert(m_nindices.size() == m_ind_types.size());
5129 lean_assert(m_ind_cnsts.size() == m_ind_types.size());
5130 lean_assert(m_params.size() == m_nparams);
5131 }
5132
5133 /** \brief Return true if declaration is recursive */
5134 bool is_rec() {
5135     for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
5136         inductive_type const &ind_type = m_ind_types[idx];
5137         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5138             expr t = constructor_type(cnstr);
5139             while (is_pi(t)) {
5140                 if (find(binding_domain(t), [&](expr const &e, unsigned) {
5141                     if (is_constant(e)) {
5142                         for (expr const &I : m_ind_cnsts)
5143                             if (const_name(I) == const_name(e))
5144                                 return true;
5145                     }
5146                     return false;
5147                 }))) {
5148                     return true;
5149                 }
5150                 t = binding_body(t);
5151             }
5152         }
5153     }
5154     return false;
5155 }
5156
5157 /** Return true if the given declarataion is reflexive.
5158
5159     Remark: We say an inductive type `T` is reflexive if it
5160     contains at least one constructor that takes as an argument a
5161     function returning `T` where `T` is another inductive datatype
5162     (possibly equal to `T`) in the same mutual declaration. */
5163 bool is_reflexive() {
5164     for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
5165         inductive_type const &ind_type = m_ind_types[idx];
5166         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5167             expr t = constructor_type(cnstr);
5168             while (is_pi(t)) {
5169                 expr arg_type = binding_domain(t);
5170                 if (is_pi(arg_type) && has_ind_occ(arg_type)) return true;
5171                 expr local = mk_local_decl_for(t);
5172                 t = instantiate(binding_body(t), local);
5173             }
5174         }
5175     }
5176     return false;
5177 }
5178
5179 /** Return list with the names of all inductive datatypes in the mutual

```



```

5180     * declaration. */
5181 names get_all_inductive_names() const {
5182     return ::lean::get_all_inductive_names(m_ind_types);
5183 }
5184
5185 /** \brief Add all datatype declarations to environment. */
5186 void declare_inductive_types() {
5187     bool rec = is_rec();
5188     bool reflexive = is_reflexive();
5189     names all = get_all_inductive_names();
5190     for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
5191         inductive_type const &ind_type = m_ind_types[idx];
5192         name const &n = ind_type.get_name();
5193         buffer<name> cnstr_names;
5194         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5195             cnstr_names.push_back(constructor_name(cnstr));
5196         }
5197         m_env.add_core(constant_info(
5198             inductive_val(n, m_lparams, ind_type.get_type(), m_nparams,
5199                 m_nindices[idx], all, names(cnstr_names), rec,
5200                 m_is_unsafe, reflexive, m_is_nested)));
5201     }
5202 }
5203
5204 /** \brief Return true iff `t` is a term of the form `I As t`
5205     where `I` is the inductive datatype at position `i` being declared and
5206     `As` are the global parameters of this declaration. */
5207 bool is_valid_ind_app(expr const &t, unsigned i) {
5208     buffer<expr> args;
5209     expr I = get_app_args(t, args);
5210     if (I != m_ind_cnsts[i] || args.size() != m_nparams + m_nindices[i])
5211         return false;
5212     for (unsigned i = 0; i < m_nparams; i++) {
5213         if (m_params[i] != args[i]) return false;
5214     }
5215     return true;
5216 }
5217
5218 /** \brief Return some(i) iff `t` is of the form `I As t` where `I` the
5219     * inductive `i`-th datatype being defined. */
5220 optional<unsigned> is_valid_ind_app(expr const &t) {
5221     for (unsigned i = 0; i < m_ind_types.size(); i++) {
5222         if (is_valid_ind_app(t, i)) return optional<unsigned>(i);
5223     }
5224     return optional<unsigned>();
5225 }
5226
5227 /** \brief Return true iff `e` is one of the inductive datatype being
5228     * declared. */
5229 bool is_ind_occ(expr const &e) {
5230     return is_constant(e) &&
5231         std::any_of(m_ind_cnsts.begin(), m_ind_cnsts.end(),
5232             [&](expr const &c) {
5233                 return const_name(e) == const_name(c);
5234             });
5235 }
5236
5237 /** \brief Return true iff `t` does not contain any occurrence of a datatype
5238     * being declared. */
5239 bool has_ind_occ(expr const &t) {
5240     return static_cast<bool>(
5241         find(t, [&](expr const &e, unsigned) { return is_ind_occ(e); }));
5242 }
5243
5244 /** \brief Return `some(d_idx)` iff `t` is a recursive argument, `d_idx` is
5245     the index of the
5246     recursive inductive datatype. Otherwise, return `none`. */
5247 optional<unsigned> is_rec_argument(expr t) {
5248     t = whnf(t);
5249     while (is_pi(t)) {

```

```

5250         expr local = mk_local_decl_for(t);
5251         t = whnf(instantiate(binding_body(t), local));
5252     }
5253     return is_valid_ind_app(t);
5254 }
5255
5256 /** \brief Check if \c t contains only positive occurrences of the inductive
5257  * datatypes being declared. */
5258 void check_positivity(expr t, name const &cnstr_name, int arg_idx) {
5259     t = whnf(t);
5260     if (!has_ind_occ(t)) {
5261         // nonrecursive argument
5262     } else if (is_pi(t)) {
5263         if (has_ind_occ(binding_domain(t)))
5264             throw kernel_exception(
5265                 m_env, sstream() << "arg #" << (arg_idx + 1) << " of '"
5266                     << cnstr_name
5267                     << "' "
5268                     << "has a non positive occurrence of the "
5269                     << "datatypes being declared");
5270         expr local = mk_local_decl_for(t);
5271         check_positivity(instantiate(binding_body(t), local), cnstr_name,
5272             arg_idx);
5273     } else if (is_valid_ind_app(t)) {
5274         // recursive argument
5275     } else {
5276         throw kernel_exception(
5277             m_env, sstream()
5278                 << "arg #" << (arg_idx + 1) << " of '" << cnstr_name
5279                 << "' "
5280                 << "contains a non valid occurrence of the "
5281                 << "datatypes being declared");
5282     }
5283 }
5284
5285 /** \brief Check whether the constructor declarations are type correct,
5286  parameters are in the expected positions, constructor fields are in
5287  acceptable universe levels, positivity constraints, and returns the
5288  expected result. */
5289 void check_constructors() {
5290     for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
5291         inductive_type const &ind_type = m_ind_types[idx];
5292         name_set found_cnstrs;
5293         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5294             name const &n = constructor_name(cnstr);
5295             if (found_cnstrs.contains(n)) {
5296                 throw kernel_exception(
5297                     m_env, sstream() << "duplicate constructor name '" << n
5298                         << "'");
5299             }
5300             found_cnstrs.insert(n);
5301             expr t = constructor_type(cnstr);
5302             m_env.check_name(n);
5303             check_no_metavar_no_fvar(m_env, n, t);
5304             tc().check(t, m_lparams);
5305             unsigned i = 0;
5306             while (is_pi(t)) {
5307                 if (i < m_nparams) {
5308                     if (!is_def_eq(binding_domain(t), get_param_type(i)))
5309                         throw kernel_exception(
5310                             m_env, sstream() << "arg #" << (i + 1)
5311                                 << " of '" << n << "' "
5312                                 << "does not match inductive "
5313                                 << "datatypes parameters'");
5314                     t = instantiate(binding_body(t), m_params[i]);
5315                 } else {
5316                     expr s = tc().ensure_type(binding_domain(t));
5317                     // the sort is ok IF
5318                     // 1- its level is <= inductive datatype level, OR
5319                     // 2- is an inductive predicate

```

```

5320         if (!(is_geq(m_result_level, sort_level(s)) ||
5321             is_zero(m_result_level))) {
5322             throw kernel_exception(
5323                 m_env,
5324                 sstream()
5325                     << "universe level of type_of(arg #"
5326                     << (i + 1) << ") "
5327                     << "of '" << n
5328                     << "' is too big for the corresponding "
5329                     << "inductive datatype");
5330         }
5331         if (!m_is_unsafe)
5332             check_positivity(binding_domain(t), n, i);
5333         expr local = mk_local_decl_for(t);
5334         t = instantiate(binding_body(t), local);
5335     }
5336     i++;
5337 }
5338 if (!is_valid_ind_app(t, idx))
5339     throw kernel_exception(
5340         m_env, sstream()
5341             << "invalid return type for '" << n << "'");
5342 }
5343 }
5344 }
5345
5346 void declare_constructors() {
5347     for (unsigned idx = 0; idx < m_ind_types.size(); idx++) {
5348         inductive_type const &ind_type = m_ind_types[idx];
5349         unsigned cidx = 0;
5350         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5351             name const &n = constructor_name(cnstr);
5352             expr const &t = constructor_type(cnstr);
5353             unsigned arity = 0;
5354             expr it = t;
5355             while (is_pi(it)) {
5356                 it = binding_body(it);
5357                 arity++;
5358             }
5359             lean_assert(arity >= m_nparams);
5360             unsigned nfields = arity - m_nparams;
5361             m_env.add_core(constant_info(
5362                 constructor_val(n, m_lparams, t, ind_type.get_name(), cidx,
5363                     m_nparams, nfields, m_is_unsafe)));
5364             cidx++;
5365         }
5366     }
5367 }
5368
5369 /** \brief Return true if recursor can only map into Prop */
5370 bool elim_only_at_universe_zero() {
5371     if (m_is_not_zero) {
5372         /* For every universe parameter assignment, the resultant universe
5373            is not 0. So, it is not an inductive predicate */
5374         return false;
5375     }
5376
5377     if (m_ind_types.size() > 1) {
5378         /* Mutually recursive inductive predicates only eliminate into Prop.
5379            */
5380         return true;
5381     }
5382
5383     unsigned num_intros = length(m_ind_types[0].get_cnstrs());
5384     if (num_intros > 1) {
5385         /* We have more than one constructor, then recursor for inductive
5386            predicate can only eliminate intro Prop. */
5387         return true;
5388     }
5389 }

```

```

5390     if (num_intros == 0) {
5391         /* empty inductive predicate (e.g., `false`) can eliminate into any
5392            * universe */
5393         return false;
5394     }
5395
5396     /* We have only one constructor, the final check is, the type of each
5397        argument that is not a parameter: 1- It must live in Prop, *OR* 2- It
5398        must occur in the return type. (this is essentially what is called a
5399        non-uniform parameter in Coq). We can justify 2 by observing that
5400        this information is not a *secret* it is part of the type. By
5401        eliminating to a non-proposition, we would not be revealing anything
5402        that is not already known. */
5403     constructor const &cstr = head(m_ind_types[0].get_cnstrs());
5404     expr type = constructor_type(cstr);
5405     unsigned i = 0;
5406     buffer<expr> to_check; /* Arguments that we must check if occur in the
5407                            result type */
5408     while (is_pi(type)) {
5409         expr fvar = mk_local_decl_for(type);
5410         if (i >= m_nparams) {
5411             expr s = tc().ensure_type(binding_domain(type));
5412             if (!is_zero(sort_level(s))) {
5413                 /* Current argument is not in Prop (i.e., condition 1
5414                    failed). We save it in to_check to be able to try
5415                    condition 2 above. */
5416                 to_check.push_back(fvar);
5417             }
5418         }
5419         type = instantiate(binding_body(type), fvar);
5420         i++;
5421     }
5422     buffer<expr> result_args;
5423     get_app_args(type, result_args);
5424     /* Check condition 2: every argument in to_check must occur in
5425        * result_args */
5426     for (expr const &arg : to_check) {
5427         if (std::find(result_args.begin(), result_args.end(), arg) ==
5428             result_args.end())
5429             return true; /* Condition 2 failed */
5430     }
5431     return false;
5432 }
5433
5434 /** \brief Initialize m_elim_level. */
5435 void init_elim_level() {
5436     if (elim_only_at_universe_zero()) {
5437         m_elim_level = mk_level_zero();
5438     } else {
5439         name u("u");
5440         int i = 1;
5441         while (std::find(m_lparams.begin(), m_lparams.end(), u) !=
5442             m_lparams.end()) {
5443             u = name("u").append_after(i);
5444             i++;
5445         }
5446         m_elim_level = mk_univ_param(u);
5447     }
5448 }
5449
5450 void init_K_target() {
5451     /* A declaration is target for K-like reduction when
5452        it has one intro, the intro has 0 arguments, and it is an inductive
5453        predicate.
5454        In the following for-loop we check if the intro rule has 0 fields. */
5455     m_K_target =
5456         m_ind_types.size() ==
5457         1 && /* It is not a mutual declaration (for simplicity, we don't
5458            gain anything by supporting K in mutual declarations. */
5459         is_zero(m_result_level) && /* It is an inductive predicate. */

```

```

5460         length(m_ind_types[0].get_cnstrs()) ==
5461         1; /* Inductive datatype has only one constructor. */
5462     if (!m_K_target) return;
5463     expr it = constructor_type(head(m_ind_types[0].get_cnstrs()));
5464     unsigned i = 0;
5465     while (is_pi(it)) {
5466         if (i < m_nparams) {
5467             it = binding_body(it);
5468         } else {
5469             /* See comment above */
5470             m_K_target = false;
5471             break;
5472         }
5473         i++;
5474     }
5475 }
5476
5477 /** \brief Given `t` of the form `I As is` where `I` is one of the inductive
5478     datatypes being defined, As are the global parameters, and is the actual
5479     indices provided to it. Return the index of `I`, and store is in the
5480     argument `indices`. */
5481 unsigned get_I_indices(expr const &t, buffer<expr> &indices) {
5482     optional<unsigned> r = is_valid_ind_app(t);
5483     lean_assert(r);
5484     buffer<expr> all_args;
5485     get_app_args(t, all_args);
5486     for (unsigned i = m_nparams; i < all_args.size(); i++)
5487         indices.push_back(all_args[i]);
5488     return *r;
5489 }
5490
5491 /** \brief Populate m_rec_infos. */
5492 void mk_rec_infos() {
5493     unsigned d_idx = 0;
5494     /* First, populate the fields, m_C, m_indices, m_major */
5495     for (inductive_type const &ind_type : m_ind_types) {
5496         rec_info info;
5497         expr t = ind_type.get_type();
5498         unsigned i = 0;
5499         while (is_pi(t)) {
5500             if (i < m_nparams) {
5501                 t = instantiate(binding_body(t), m_params[i]);
5502             } else {
5503                 expr idx = mk_local_decl_for(t);
5504                 info.m_indices.push_back(idx);
5505                 t = instantiate(binding_body(t), idx);
5506             }
5507             i++;
5508         }
5509         info.m_major = mk_local_decl(
5510             "t",
5511             mk_app(mk_app(m_ind_cnsts[d_idx], m_params), info.m_indices));
5512         expr C_ty = mk_sort(m_elim_level);
5513         C_ty = mk_pi(info.m_major, C_ty);
5514         C_ty = mk_pi(info.m_indices, C_ty);
5515         name C_name("motive");
5516         if (m_ind_types.size() > 1)
5517             C_name = name(C_name).append_after(d_idx + 1);
5518         info.m_C = mk_local_decl(C_name, C_ty);
5519         m_rec_infos.push_back(info);
5520         d_idx++;
5521     }
5522     /* First, populate the field m_minors */
5523     unsigned minor_idx = 1;
5524     d_idx = 0;
5525     for (inductive_type const &ind_type : m_ind_types) {
5526         name ind_type_name = ind_type.get_name();
5527         for (constructor const &cnstr : ind_type.get_cnstrs()) {
5528             buffer<expr> b_u; // nonrec and rec args;
5529             buffer<expr> u; // rec args

```

```

5530     buffer<expr> v;    // inductive args
5531     name cnstr_name = constructor_name(cnstr);
5532     expr t = constructor_type(cnstr);
5533     unsigned i = 0;
5534     while (is_pi(t)) {
5535         if (i < m_nparams) {
5536             t = instantiate(binding_body(t), m_params[i]);
5537         } else {
5538             expr l = mk_local_decl_for(t);
5539             b_u.push_back(l);
5540             if (is_rec_argument(binding_domain(t))) u.push_back(l);
5541             t = instantiate(binding_body(t), l);
5542         }
5543         i++;
5544     }
5545     buffer<expr> it_indices;
5546     unsigned it_idx = get_I_indices(t, it_indices);
5547     expr C_app = mk_app(m_rec_infos[it_idx].m_C, it_indices);
5548     expr intro_app = mk_app(
5549         mk_app(mk_constant(cnstr_name, m_levels), m_params), b_u);
5550     C_app = mk_app(C_app, intro_app);
5551     /* populate v using u */
5552     for (unsigned i = 0; i < u.size(); i++) {
5553         expr u_i = u[i];
5554         expr u_i_ty = whnf(infer_type(u_i));
5555         buffer<expr> xs;
5556         while (is_pi(u_i_ty)) {
5557             expr x = mk_local_decl_for(u_i_ty);
5558             xs.push_back(x);
5559             u_i_ty = whnf(instantiate(binding_body(u_i_ty), x));
5560         }
5561         buffer<expr> it_indices;
5562         unsigned it_idx = get_I_indices(u_i_ty, it_indices);
5563         expr C_app = mk_app(m_rec_infos[it_idx].m_C, it_indices);
5564         expr u_app = mk_app(u_i, xs);
5565         C_app = mk_app(C_app, u_app);
5566         expr v_i_ty = mk_pi(xs, C_app);
5567         expr v_i = mk_local_decl(name("v").append_after(i), v_i_ty,
5568                                 binder_info());
5569         v.push_back(v_i);
5570     }
5571     expr minor_ty = mk_pi(b_u, mk_pi(v, C_app));
5572     name minor_name =
5573         cnstr_name.replace_prefix(ind_type_name, name());
5574     expr minor = mk_local_decl(minor_name, minor_ty);
5575     m_rec_infos[d_idx].m_minors.push_back(minor);
5576     minor_idx++;
5577 }
5578 d_idx++;
5579 }
5580 }
5581
5582 /** \brief Return the levels for the recursor. */
5583 levels get_rec_levels() {
5584     if (is_param(m_elim_level))
5585         return levels(m_elim_level, m_levels);
5586     else
5587         return m_levels;
5588 }
5589
5590 /** \brief Return the level parameter names for the recursor. */
5591 names get_rec_lparams() {
5592     if (is_param(m_elim_level))
5593         return names(param_id(m_elim_level), m_lparams);
5594     else
5595         return m_lparams;
5596 }
5597
5598 /** \brief Store all type formers in `Cs` */
5599 void collect-Cs(buffer<expr> &Cs) {

```



```

5600     for (unsigned i = 0; i < m_ind_types.size(); i++)
5601         Cs.push_back(m_rec_infos[i].m_C);
5602 }
5603
5604 /** \brief Store all minor premises in `ms`. */
5605 void collect_minor_premises(buffer<expr> &ms) {
5606     for (unsigned i = 0; i < m_ind_types.size(); i++)
5607         ms.append(m_rec_infos[i].m_minors);
5608 }
5609
5610 recursor_rules mk_rec_rules(unsigned d_idx, buffer<expr> const &Cs,
5611                             buffer<expr> const &minors,
5612                             unsigned &minor_idx) {
5613     inductive_type const &d = m_ind_types[d_idx];
5614     levels lvls = get_rec_levels();
5615     buffer<recursor_rule> rules;
5616     for (constructor const &cnstr : d.get_cnstrs()) {
5617         buffer<expr> b_u;
5618         buffer<expr> u;
5619         expr t = constructor_type(cnstr);
5620         unsigned i = 0;
5621         while (is_pi(t)) {
5622             if (i < m_nparams) {
5623                 t = instantiate(binding_body(t), m_params[i]);
5624             } else {
5625                 expr l = mk_local_decl_for(t);
5626                 b_u.push_back(l);
5627                 if (is_rec_argument(binding_domain(t))) u.push_back(l);
5628                 t = instantiate(binding_body(t), l);
5629             }
5630             i++;
5631         }
5632         buffer<expr> v;
5633         for (unsigned i = 0; i < u.size(); i++) {
5634             expr u_i = u[i];
5635             expr u_i_ty = whnf(infer_type(u_i));
5636             buffer<expr> xs;
5637             while (is_pi(u_i_ty)) {
5638                 expr x = mk_local_decl_for(u_i_ty);
5639                 xs.push_back(x);
5640                 u_i_ty = whnf(instantiate(binding_body(u_i_ty), x));
5641             }
5642             buffer<expr> it_indices;
5643             unsigned it_idx = get_I_indices(u_i_ty, it_indices);
5644             name rec_name = mk_rec_name(m_ind_types[it_idx].get_name());
5645             expr rec_app = mk_constant(rec_name, lvls);
5646             rec_app = mk_app(
5647                 mk_app(
5648                     mk_app(mk_app(rec_app, m_params), Cs), minors),
5649                     it_indices,
5650                     mk_app(u_i, xs));
5651             v.push_back(mk_lambda(xs, rec_app));
5652         }
5653         expr e_app = mk_app(mk_app(minors[minor_idx], b_u), v);
5654         expr comp_rhs = mk_lambda(
5655             m_params,
5656             mk_lambda(Cs, mk_lambda(minors, mk_lambda(b_u, e_app))));
5657         rules.push_back(
5658             recursor_rule(constructor_name(cnstr), b_u.size(), comp_rhs));
5659         minor_idx++;
5660     }
5661     return recursor_rules(rules);
5662 }
5663
5664 /** \brief Declare recursors. */
5665 void declare_recurors() {
5666     buffer<expr> Cs;
5667     collect-Cs(Cs);
5668     buffer<expr> minors;
5669     collect_minor_premises(minors);

```

```

5670     unsigned nminors = minors.size();
5671     unsigned nmotives = Cs.size();
5672     names all = get_all_inductive_names();
5673     unsigned minor_idx = 0;
5674     for (unsigned d_idx = 0; d_idx < m_ind_types.size(); d_idx++) {
5675         rec_info const &info = m_rec_infos[d_idx];
5676         expr C_app = mk_app(mk_app(info.m_C, info.m_indices), info.m_major);
5677         expr rec_ty = mk_pi(info.m_major, C_app);
5678         rec_ty = mk_pi(info.m_indices, rec_ty);
5679         rec_ty = mk_pi(minors, rec_ty);
5680         rec_ty = mk_pi(Cs, rec_ty);
5681         rec_ty = mk_pi(m_params, rec_ty);
5682         rec_ty = infer_implicit(rec_ty, true /* strict */);
5683         recursor_rules rules = mk_rec_rules(d_idx, Cs, minors, minor_idx);
5684         name rec_name = mk_rec_name(m_ind_types[d_idx].get_name());
5685         names rec_lparams = get_rec_lparams();
5686         m_env.add_core(constant_info(
5687             recursor_val(rec_name, rec_lparams, rec_ty, all, m_nparams,
5688                 m_nindices[d_idx], nmotives, nminors, rules,
5689                 m_K_target, m_is_unsafe)));
5690     }
5691 }
5692
5693 environment operator()() {
5694     m_env.check_duplicated_univ_params(m_lparams);
5695     check_inductive_types();
5696     declare_inductive_types();
5697     check_constructors();
5698     declare_constructors();
5699     init_elim_level();
5700     init_K_target();
5701     mk_rec_infos();
5702     declare_recursors();
5703     return m_env;
5704 }
5705 };
5706
5707 static name *g_nested = nullptr;
5708 static name *g_nested_fresh = nullptr;
5709
5710 /* Result produced by elim_nested_inductive_fn */
5711 struct elim_nested_inductive_result {
5712     name_generator m_ngen;
5713     buffer<expr> m_params;
5714     name_map<expr> m_aux2nested; /* mapping from auxiliary type to nested
5715                                inductive type. */
5716     declaration m_aux_decl;
5717
5718     elim_nested_inductive_result(name_generator const &ngen,
5719                                buffer<expr> const &params,
5720                                buffer<pair<expr, name>> const &nested_aux,
5721                                declaration const &d) {
5722         : m_ngen(ngen), m_params(params), m_aux_decl(d) {
5723             for (pair<expr, name> const &p : nested_aux) {
5724                 m_aux2nested.insert(p.second, p.first);
5725             }
5726         }
5727
5728     /* If `c` is an constructor name associated with an auxiliary inductive
5729        type, then return the
5730        nested inductive associated with it and the name of its inductive type.
5731        Return none. */
5732     optional<pair<expr, name>> get_nested_if_aux_constructor(
5733         environment const &aux_env, name const &c) const {
5734         optional<constant_info> info = aux_env.find(c);
5735         if (!info || !info->is_constructor())
5736             return optional<pair<expr, name>>();
5737         name auxI_name = info->to_constructor_val().get_induct();
5738         expr const *nested = m_aux2nested.find(auxI_name);
5739         if (!nested) return optional<pair<expr, name>>();

```

```

5740     return optional<pair<expr, name>>(*nested, auxI_name);
5741 }
5742
5743 name restore_constructor_name(environment const &aux_env,
5744                               name const &cnstr_name) const {
5745     optional<pair<expr, name>> p =
5746         get_nested_if_aux_constructor(aux_env, cnstr_name);
5747     lean_assert(p);
5748     expr const &I = get_app_fn(p->first);
5749     lean_assert(is_constant(I));
5750     return cnstr_name.replace_prefix(p->second, const_name(I));
5751 }
5752
5753 expr restore_nested(
5754     expr e, environment const &aux_env,
5755     name_map<name> const &aux_rec_name_map = name_map<name>()) {
5756     local_ctx lctx;
5757     buffer<expr> As;
5758     bool pi = is_pi(e);
5759     for (unsigned i = 0; i < m_params.size(); i++) {
5760         lean_assert(is_pi(e) || is_lambda(e));
5761         As.push_back(lctx.mk_local_decl(
5762             m_ngen, binding_name(e), binding_domain(e), binding_info(e)));
5763         e = instantiate(binding_body(e), As.back());
5764     }
5765     e = replace(e, [&](expr const &t, unsigned) {
5766         if (is_constant(t)) {
5767             if (name const *rec_name =
5768                 aux_rec_name_map.find(const_name(t))) {
5769                 return some_expr(mk_constant(*rec_name, const_levels(t)));
5770             }
5771         }
5772         expr const &fn = get_app_fn(t);
5773         if (is_constant(fn)) {
5774             if (expr const *nested = m_aux2nested.find(const_name(fn))) {
5775                 buffer<expr> args;
5776                 get_app_args(t, args);
5777                 lean_assert(args.size() >= m_params.size());
5778                 expr new_t = instantiate_rev(
5779                     abstract(*nested, m_params.size(), m_params.data()),
5780                     As.size(), As.data());
5781                 return some_expr(mk_app(new_t,
5782                                         args.size() - m_params.size(),
5783                                         args.data() + m_params.size()));
5784             }
5785             if (optional<pair<expr, name>> r =
5786                 get_nested_if_aux_constructor(aux_env,
5787                                                 const_name(fn))) {
5788                 expr nested = r->first;
5789                 name auxI_name = r->second;
5790                 /* `t` is a constructor-application of an auxiliary
5791                  * inductive type */
5792                 buffer<expr> args;
5793                 get_app_args(t, args);
5794                 lean_assert(args.size() >= m_params.size());
5795                 expr new_nested = instantiate_rev(
5796                     abstract(nested, m_params.size(), m_params.data()),
5797                     As.size(), As.data());
5798                 buffer<expr> I_args;
5799                 expr I = get_app_args(new_nested, I_args);
5800                 lean_assert(is_constant(I));
5801                 name new_fn_name =
5802                     const_name(fn).replace_prefix(auxI_name, const_name(I));
5803                 expr new_fn = mk_constant(new_fn_name, const_levels(I));
5804                 expr new_t = mk_app(mk_app(new_fn, I_args),
5805                                     args.size() - m_params.size(),
5806                                     args.data() + m_params.size());
5807                 return some_expr(new_t);
5808             }
5809         }

```

```

5810         return none_expr();
5811     });
5812     return pi ? lctx.mk_pi(As, e) : lctx.mk_lambda(As, e);
5813 }
5814 };
5815
5816 /* Eliminate nested inductive datatypes by creating a new (auxiliary)
5817    declaration which contains and inductive types in `d` and copies of the
5818    nested inductive datatypes used in `d`. For each nested occurrence `I Ds is`
5819    where `I` is a nested inductive datatype and `Ds` are the parametric
5820    arguments and `is` the indices, we create an auxiliary type `Iaux` in the
5821    (mutual) inductive declaration `d`, and replace `I Ds is` with `Iaux As is`
5822    where `As` are `d`'s parameters. Moreover, we add the pair `(I Ds, Iaux)` to
5823    `nested_aux`.
5824
5825    Note that, `As` and `Ds` may have a different sizes. */
5826 struct elim_nested_inductive_fn {
5827     environment const &m_env;
5828     declaration const &m_d;
5829     name_generator m_ngen;
5830     local_ctx m_params_lctx;
5831     buffer<expr> m_params;
5832     buffer<pair<expr, name>>
5833         m_nested_aux; /* The expressions stored here contains free vars in
5834                        `m_params` */
5835     levels m_lvls;
5836     buffer<inductive_type> m_new_types;
5837     unsigned m_next_idx{1};
5838
5839     elim_nested_inductive_fn(environment const &env, declaration const &d)
5840         : m_env(env), m_d(d), m_ngen(*g_nested_fresh) {
5841         m_lvls = lparams_to_levels(inductive_decl(m_d).get_lparams());
5842     }
5843
5844     name mk_unique_name(name const &n) {
5845         while (true) {
5846             name r = n.append_after(m_next_idx);
5847             m_next_idx++;
5848             if (!m_env.find(r)) return r;
5849         }
5850     }
5851
5852     void throw_ill_formed() {
5853         throw kernel_exception(
5854             m_env, "invalid nested inductive datatype, ill-formed declaration");
5855     }
5856
5857     expr replace_params(expr const &e, buffer<expr> const &As) {
5858         lean_assert(m_params.size() == As.size());
5859         return instantiate_rev(abstract(e, As.size(), As.data()),
5860                                m_params.size(), m_params.data());
5861     }
5862
5863     /* IF `e` is of the form `I Ds is` where
5864        1) `I` is a nested inductive datatype (i.e., a previously declared
5865        inductive datatype), 2) the parametric arguments `Ds` do not contain
5866        loose bound variables, and do contain inductive datatypes in
5867        `m_new_types` THEN return the `inductive_val` in the `constant_info`
5868        associated with `I`. Otherwise, return none. */
5869     optional<inductive_val> is_nested_inductive_app(expr const &e) {
5870         if (!is_app(e)) return optional<inductive_val>();
5871         expr const &fn = get_app_fn(e);
5872         if (!is_constant(fn)) return optional<inductive_val>();
5873         optional<constant_info> info = m_env.find(const_name(fn));
5874         if (!info || !info->is_inductive()) return optional<inductive_val>();
5875         buffer<expr> args;
5876         get_app_args(e, args);
5877         unsigned nparams = info->to_inductive_val().get_nparams();
5878         if (nparams > args.size()) return optional<inductive_val>();
5879         bool is_nested = false;

```

```

5880     bool loose_bvars = false;
5881     for (unsigned i = 0; i < nparams; i++) {
5882         if (has_loose_bvars(args[i])) {
5883             loose_bvars = true;
5884         }
5885         if (find(args[i], [&](expr const &t, unsigned) {
5886             if (is_constant(t)) {
5887                 for (inductive_type const &ind_type : m_new_types) {
5888                     if (const_name(t) == ind_type.get_name())
5889                         return true;
5890                 }
5891             }
5892             return false;
5893         }))) {
5894             is_nested = true;
5895         }
5896     }
5897     if (!is_nested) return optional<inductive_val>();
5898     if (loose_bvars)
5899         throw kernel_exception(
5900             m_env, sstream() << "invalid nested inductive datatype '"
5901                             << const_name(fn)
5902                             << "', nested inductive datatypes parameters '"
5903                             << "cannot contain local variables.");
5904     return optional<inductive_val>(info->to_inductive_val());
5905 }
5906
5907 expr instantiate_pi_params(expr e, unsigned nparams, expr const *params) {
5908     for (unsigned i = 0; i < nparams; i++) {
5909         if (!is_pi(e)) throw_ill_formed();
5910         e = binding_body(e);
5911     }
5912     return instantiate_rev(e, nparams, params);
5913 }
5914
5915 /* If `e` is a nested occurrence `I Ds is`, return `Iaux As is` */
5916 optional<expr> replace_if_nested(local_ctx const &lctx,
5917     buffer<expr> const &As, expr const &e) {
5918     optional<inductive_val> I_val = is_nested_inductive_app(e);
5919     if (!I_val) return none_expr();
5920     /* `e` is of the form `I As is` where `As` are the parameters and `is`
5921      * the indices */
5922     buffer<expr> args;
5923     expr const &fn = get_app_args(e, args);
5924     name const &I_name = const_name(fn);
5925     levels const &I_lvls = const_levels(fn);
5926     lean_assert(I_val->get_nparams() <= args.size());
5927     unsigned I_nparams = I_val->get_nparams();
5928     expr IAs = mk_app(fn, I_nparams, args.data()); /* `I As` */
5929     /* Check whether we have already created an auxiliary inductive_type for
5930      * `I As` */
5931     optional<name> auxI_name;
5932     /* Replace `As` with `m_params` before searching at `m_nested_aux`.
5933      * We need this step because we re-create parameters for each
5934      * constructor with the correct binding info */
5935     expr Iparams = replace_params(IAs, As);
5936     for (pair<expr, name> const &p : m_nested_aux) {
5937         /* Remark: we could have used `is_def_eq` here instead of structural
5938          * equality. It is probably not needed, but if one day we decide to
5939          * do it, we have to populate an auxiliary environment with the
5940          * inductive datatypes we are defining since `p.first` and `Iparams`
5941          * contain references to them. */
5942         if (p.first == Iparams) {
5943             auxI_name = p.second;
5944             break;
5945         }
5946     }
5947     if (auxI_name) {
5948         expr auxI = mk_constant(*auxI_name, m_lvls);
5949         auxI = mk_app(auxI, As);

```

```

5950         return some_expr(
5951             mk_app(auxI, args.size() - I_nparams, args.data() + I_nparams));
5952     } else {
5953         optional<expr> result;
5954         /* We should copy all inductive datatypes `J` in the mutual
5955            declaration containing `I` to the `m_new_types` mutual
5956            declaration as new auxiliary types. */
5957         for (name const &J_name : I_val->get_all()) {
5958             constant_info J_info = m_env.get(J_name);
5959             lean_assert(J_info.is_inductive());
5960             expr J = mk_constant(J_name, I_lvls);
5961             expr JAs = mk_app(J, I_nparams, args.data());
5962             name auxJ_name = mk_unique_name(*g_nested + J_name);
5963             expr auxJ_type = instantiate_lparams(
5964                 J_info.get_type(), J_info.get_lparams(), I_lvls);
5965             auxJ_type =
5966                 instantiate_pi_params(auxJ_type, I_nparams, args.data());
5967             auxJ_type = lctx.mk_pi(As, auxJ_type);
5968             m_nested_aux.push_back(
5969                 mk_pair(replace_params(JAs, As), auxJ_name));
5970             if (J_name == I_name) {
5971                 /* Create result */
5972                 expr auxI = mk_constant(auxJ_name, m_lvls);
5973                 auxI = mk_app(auxI, As);
5974                 result = mk_app(auxI, args.size() - I_nparams,
5975                     args.data() + I_nparams);
5976             }
5977             buffer<constructor> auxJ_constructors;
5978             for (name const &J_cnstr_name :
5979                 J_info.to_inductive_val().get_cnstrs()) {
5980                 constant_info J_cnstr_info = m_env.get(J_cnstr_name);
5981                 name auxJ_cnstr_name =
5982                     J_cnstr_name.replace_prefix(J_name, auxJ_name);
5983                 /* auxJ_cnstr_type still has references to `J`, this will be
5984                    * fixed later when we process it. */
5985                 expr auxJ_cnstr_type =
5986                     instantiate_lparams(J_cnstr_info.get_type(),
5987                         J_cnstr_info.get_lparams(), I_lvls);
5988                 auxJ_cnstr_type = instantiate_pi_params(
5989                     auxJ_cnstr_type, I_nparams, args.data());
5990                 auxJ_cnstr_type = lctx.mk_pi(As, auxJ_cnstr_type);
5991                 auxJ_constructors.push_back(
5992                     constructor(auxJ_cnstr_name, auxJ_cnstr_type));
5993             }
5994             m_new_types.push_back(inductive_type(
5995                 auxJ_name, auxJ_type, constructors(auxJ_constructors)));
5996         }
5997         lean_assert(result);
5998         return result;
5999     }
6000 }
6001
6002 /* Replace all nested inductive datatype occurrences in `e`. */
6003 expr replace_all_nested(local_ctx const &lctx, buffer<expr> const &As,
6004     expr const &e) {
6005     return replace(e, [&](expr const &e, unsigned) {
6006         return replace_if_nested(lctx, As, e);
6007     });
6008 }
6009
6010 expr get_params(expr type, unsigned nparams, local_ctx &lctx,
6011     buffer<expr> &params) {
6012     lean_assert(params.empty());
6013     for (unsigned i = 0; i < nparams; i++) {
6014         if (!is_pi(type))
6015             throw kernel_exception(
6016                 m_env,
6017                 "invalid inductive datatype declaration, incorrect number "
6018                 "of parameters");
6019         params.push_back(lctx.mk_local_decl(mngen, binding_name(type),

```



```

6020                                     binding_domain(type),
6021                                     binding_info(type));
6022     type = instantiate(binding_body(type), params.back());
6023 }
6024 return type;
6025 }
6026
6027 elim_nested_inductive_result operator()() {
6028     inductive_decl ind_d(m_d);
6029     if (!ind_d.get_nparams().is_small()) throw_ill_formed();
6030     unsigned d_nparams = ind_d.get_nparams().get_small_value();
6031     to_buffer(ind_d.get_types(), m_new_types);
6032     if (m_new_types.size() == 0)
6033         throw kernel_exception(
6034             m_env,
6035             "invalid empty (mutual) inductive datatype declaration, it "
6036             "must contain at least one inductive type.");
6037     /* initialize m_params and m_params_lctx */
6038     get_params(m_new_types[0].get_type(), d_nparams, m_params_lctx,
6039         m_params);
6040     unsigned qhead = 0;
6041     /* Main elimination loop. */
6042     while (qhead < m_new_types.size()) {
6043         inductive_type ind_type = m_new_types[qhead];
6044         buffer<constructor> new_cnstrs;
6045         for (constructor cnstr : ind_type.get_cnstrs()) {
6046             expr cnstr_type = constructor_type(cnstr);
6047             local_ctx lctx;
6048             buffer<expr> As;
6049             /* Consume parameters.
6050
6051             We (re-)create the parameters for each constructor because we
6052             want to preserve the binding_info. */
6053             cnstr_type = get_params(cnstr_type, d_nparams, lctx, As);
6054             lean_assert(As.size() == d_nparams);
6055             expr new_cnstr_type = replace_all_nested(lctx, As, cnstr_type);
6056             new_cnstr_type = lctx.mk_pi(As, new_cnstr_type);
6057             new_cnstrs.push_back(
6058                 constructor(constructor_name(cnstr), new_cnstr_type));
6059         }
6060         m_new_types[qhead] =
6061             inductive_type(ind_type.get_name(), ind_type.get_type(),
6062                 constructors(new_cnstrs));
6063         qhead++;
6064     }
6065     declaration aux_decl =
6066         mk_inductive_decl(ind_d.get_lparams(), ind_d.get_nparams(),
6067             inductive_types(m_new_types), ind_d.is_unsafe());
6068     return elim_nested_inductive_result(mngen, m_params, m_nested_aux,
6069         aux_decl);
6070 }
6071 };
6072
6073 /* Given the auxiliary environment `aux_env` generated by processing the
6074 auxiliary mutual declaration, and the original declaration `d`. This function
6075 return a pair `(aux_rec_names, aux_rec_name_map)` where `aux_rec_names`
6076 contains the recursor names associated to auxiliary inductive types used to
6077 eliminated nested inductive occurrences.
6078 The mapping `aux_rec_name_map` contains an entry `(aux_rec_name -> rec_name)`
6079 for each element in `aux_rec_names`. It provides the new names for these
6080 recursors.
6081
6082 We compute the new recursor names using the first inductive datatype in the
6083 original declaration `d`, and the suffice `.rec_<idx>`. */
6084 static pair<names, name_map<name>> mk_aux_rec_name_map(
6085     environment const &aux_env, inductive_decl const &d) {
6086     unsigned ntypes = length(d.get_types());
6087     lean_assert(ntypes > 0);
6088     inductive_type const &main_type = head(d.get_types());
6089     name const &main_name = main_type.get_name();

```

```

6090     constant_info main_info = aux_env.get(main_name);
6091     names const &all_names = main_info.to_inductive_val().get_all();
6092     /* This function is only called if we have created auxiliary inductive types
6093        when eliminating the nested inductives. */
6094     lean_assert(length(all_names) > ntypes);
6095     /* Remark: we use the `main_name` to declare the auxiliary recursors as:
6096        <main_name>.rec_1, <main_name>.rec_2, ... This is a little bit
6097        asymmetrical if `d` is a mutual declaration, but it makes sure we have
6098        simple names. */
6099     buffer<name> old_rec_names;
6100     name_map<name> rec_map;
6101     unsigned i = 0;
6102     unsigned next_idx = 1;
6103     for (name const &ind_name : all_names) {
6104         if (i >= ntypes) {
6105             old_rec_names.push_back(mk_rec_name(ind_name));
6106             name new_rec_name = mk_rec_name(main_name).append_after(next_idx);
6107             next_idx++;
6108             rec_map.insert(old_rec_names.back(), new_rec_name);
6109         }
6110         i++;
6111     }
6112     return mk_pair(names(old_rec_names), rec_map);
6113 }
6114
6115 environment environment::add_inductive(declaration const &d) const {
6116     elim_nested_inductive_result res = elim_nested_inductive_fn(*this, d)();
6117     bool is_nested = !res.m_aux2nested.empty();
6118     environment aux_env =
6119         add_inductive_fn(*this, inductive_decl(res.m_aux_decl), is_nested)();
6120     if (!is_nested) {
6121         /* `d` did not contain nested inductive types. */
6122         return aux_env;
6123     } else {
6124         /* Restore nested inductives. */
6125         inductive_decl ind_d(d);
6126         names all_ind_names = get_all_inductive_names(ind_d);
6127         names aux_rec_names;
6128         name_map<name> aux_rec_name_map;
6129         std::tie(aux_rec_names, aux_rec_name_map) =
6130             mk_aux_rec_name_map(aux_env, d);
6131         environment new_env = *this;
6132         auto process_rec = [&](name const &rec_name) {
6133             name new_rec_name = rec_name;
6134             if (name const *new_name = aux_rec_name_map.find(rec_name))
6135                 new_rec_name = *new_name;
6136             constant_info rec_info = aux_env.get(rec_name);
6137             expr new_rec_type = res.restore_nested(rec_info.get_type(), aux_env,
6138                                                     aux_rec_name_map);
6139             recursor_val rec_val = rec_info.to_recursor_val();
6140             buffer<recursor_rule> new_rules;
6141             for (recursor_rule const &rule : rec_val.get_rules()) {
6142                 expr new_rhs = res.restore_nested(rule.get_rhs(), aux_env,
6143                                                     aux_rec_name_map);
6144                 name cnstr_name = rule.get_cnstr();
6145                 name new_cnstr_name = cnstr_name;
6146                 if (new_rec_name != rec_name) {
6147                     /* We need to fix the constructor name */
6148                     new_cnstr_name =
6149                         res.restore_constructor_name(aux_env, cnstr_name);
6150                 }
6151                 new_rules.push_back(
6152                     recursor_rule(new_cnstr_name, rule.get_nfields(), new_rhs));
6153             }
6154             new_env.check_name(new_rec_name);
6155             new_env.add_core(constant_info(
6156                 recursor_val(new_rec_name, rec_info.get_lparams(), new_rec_type,
6157                             all_ind_names, rec_val.get_nparams(),
6158                             rec_val.get_nindices(), rec_val.get_nmotives(),
6159                             rec_val.get_nminors(), recursor_rules(new_rules),

```

```

6160         rec_val.is_k(), rec_val.is_unsafe())));
6161     };
6162     for (inductive_type const &ind_type : ind_d.get_types()) {
6163         constant_info ind_info = aux_env.get(ind_type.get_name());
6164         inductive_val ind_val = ind_info.to_inductive_val();
6165         /* We just need to "fix" the `all` fields for ind_info.
6166
6167         Remark: if we decide to store the recursor names, we will also
6168         need to fix it. */
6169         new_env.add_core(constant_info(inductive_val(
6170             ind_info.get_name(), ind_info.get_lparams(),
6171             ind_info.get_type(), ind_val.get_nparams(),
6172             ind_val.get_nindices(), all_ind_names, ind_val.get_cnstrs(),
6173             ind_val.is_rec(), ind_val.is_unsafe(), ind_val.is_reflexive(),
6174             ind_val.is_nested())));
6175         for (name const &cnstr_name : ind_val.get_cnstrs()) {
6176             constant_info cnstr_info = aux_env.get(cnstr_name);
6177             constructor_val cnstr_val = cnstr_info.to_constructor_val();
6178             expr new_type =
6179                 res.restore_nested(cnstr_info.get_type(), aux_env);
6180             new_env.add_core(constant_info(constructor_val(
6181                 cnstr_info.get_name(), cnstr_info.get_lparams(), new_type,
6182                 cnstr_val.get_induct(), cnstr_val.get_cidx(),
6183                 cnstr_val.get_nparams(), cnstr_val.get_nfields(),
6184                 cnstr_val.is_unsafe())));
6185         }
6186         process_rec(mk_rec_name(ind_type.get_name()));
6187     }
6188     for (name const &aux_rec : aux_rec_names) {
6189         process_rec(aux_rec);
6190     }
6191     return new_env;
6192 }
6193 }
6194
6195 static expr *g_nat_zero = nullptr;
6196 static expr *g_nat_succ = nullptr;
6197 static expr *g_string_mk = nullptr;
6198 static expr *g_list_cons_char = nullptr;
6199 static expr *g_list_nil_char = nullptr;
6200 static expr *g_char_of_nat = nullptr;
6201
6202 expr nat_lit_to_constructor(expr const &e) {
6203     lean_assert(is_nat_lit(e));
6204     nat const &v = lit_value(e).get_nat();
6205     if (v == 0u)
6206         return *g_nat_zero;
6207     else
6208         return mk_app(*g_nat_succ, mk_lit(literal(v - nat(1))));
6209 }
6210
6211 expr string_lit_to_constructor(expr const &e) {
6212     lean_assert(is_string_lit(e));
6213     string_ref const &s = lit_value(e).get_string();
6214     std::vector<unsigned> cs;
6215     utf8_decode(s.to_std_string(), cs);
6216     expr r = *g_list_nil_char;
6217     unsigned i = cs.size();
6218     while (i > 0) {
6219         i--;
6220         r = mk_app(*g_list_cons_char,
6221             mk_app(*g_char_of_nat, mk_lit(literal(cs[i]))), r);
6222     }
6223     return mk_app(*g_string_mk, r);
6224 }
6225
6226 void initialize_inductive() {
6227     g_nested = new name("_nested");
6228     mark_persistent(g_nested->raw());
6229     g_ind_fresh = new name("_ind_fresh");

```

```

6230     mark_persistent(g_ind_fresh->raw());
6231     g_nested_fresh = new name("_nested_fresh");
6232     mark_persistent(g_nested_fresh->raw());
6233     g_nat_zero = new expr(mk_constant(name{"Nat", "zero"}));
6234     mark_persistent(g_nat_zero->raw());
6235     g_nat_succ = new expr(mk_constant(name{"Nat", "succ"}));
6236     mark_persistent(g_nat_succ->raw());
6237     g_string_mk = new expr(mk_constant(name{"String", "mk"}));
6238     mark_persistent(g_string_mk->raw());
6239     expr char_type = mk_constant(name{"Char"});
6240     g_list_cons_char = new expr(
6241         mk_app(mk_constant(name{"List", "cons"}, {level()}), char_type));
6242     mark_persistent(g_list_cons_char->raw());
6243     g_list_nil_char = new expr(
6244         mk_app(mk_constant(name{"List", "nil"}, {level()}), char_type));
6245     mark_persistent(g_list_nil_char->raw());
6246     g_char_of_nat = new expr(mk_constant(name{"Char", "ofNat"}));
6247     mark_persistent(g_char_of_nat->raw());
6248     register_name_generator_prefix(*g_ind_fresh);
6249     register_name_generator_prefix(*g_nested_fresh);
6250 }
6251
6252 void finalize_inductive() {
6253     delete g_nested;
6254     delete g_ind_fresh;
6255     delete g_nested_fresh;
6256     delete g_nat_succ;
6257     delete g_nat_zero;
6258     delete g_string_mk;
6259     delete g_list_cons_char;
6260     delete g_list_nil_char;
6261 }
6262 } // namespace lean
6263 // ::::::::::::::
6264 // init_module.cpp
6265 // ::::::::::::::
6266 /*
6267 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
6268 Released under Apache 2.0 license as described in the file LICENSE.
6269
6270 Author: Leonardo de Moura
6271 */
6272 #include "kernel/declaration.h"
6273 #include "kernel/environment.h"
6274 #include "kernel/expr.h"
6275 #include "kernel/inductive.h"
6276 #include "kernel/level.h"
6277 #include "kernel/local_ctx.h"
6278 #include "kernel/quot.h"
6279 #include "kernel/type_checker.h"
6280
6281 namespace lean {
6282 void initialize_kernel_module() {
6283     initialize_level();
6284     initialize_expr();
6285     initialize_declaration();
6286     initialize_type_checker();
6287     initialize_environment();
6288     initialize_local_ctx();
6289     initialize_inductive();
6290     initialize_quot();
6291 }
6292
6293 void finalize_kernel_module() {
6294     finalize_quot();
6295     finalize_inductive();
6296     finalize_local_ctx();
6297     finalize_environment();
6298     finalize_type_checker();
6299     finalize_declaration();

```

```

6300     finalize_expr();
6301     finalize_level();
6302 }
6303 } // namespace lean
6304 // ::::::::::::::
6305 // instantiate.cpp
6306 // ::::::::::::::
6307 /*
6308 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
6309 Released under Apache 2.0 license as described in the file LICENSE.
6310
6311 Author: Leonardo de Moura
6312 */
6313 #include <algorithm>
6314 #include <limits>
6315
6316 #include "kernel/declaration.h"
6317 #include "kernel/instantiate.h"
6318 #include "kernel/replace_fn.h"
6319
6320 namespace lean {
6321 expr instantiate(expr const &a, unsigned s, unsigned n, expr const *subst) {
6322     if (s >= get_loose_bvar_range(a) || n == 0) return a;
6323     return replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
6324         unsigned s1 = s + offset;
6325         if (s1 < s)
6326             return some_expr(m); // overflow, vidx can't be >= max unsigned
6327         if (s1 >= get_loose_bvar_range(m))
6328             return some_expr(m); // expression m does not contain loose bound
6329                                     // variables with idx >= s1
6330         if (is_bvar(m)) {
6331             nat const &vidx = bvar_idx(m);
6332             if (vidx >= s1) {
6333                 unsigned h = s1 + n;
6334                 if (h < s1 /* overflow, h is bigger than any vidx */ ||
6335                     vidx < h) {
6336                     return some_expr(lift_loose_bvars(
6337                         subst[vidx.get_small_value() - s1], offset));
6338                 } else {
6339                     return some_expr(mk_bvar(vidx - nat(n)));
6340                 }
6341             }
6342         }
6343         return none_expr();
6344     });
6345 }
6346
6347 expr instantiate(expr const &e, unsigned n, expr const *s) {
6348     return instantiate(e, 0, n, s);
6349 }
6350 expr instantiate(expr const &e, std::initializer_list<expr> const &l) {
6351     return instantiate(e, l.size(), l.begin());
6352 }
6353 expr instantiate(expr const &e, unsigned i, expr const &s) {
6354     return instantiate(e, i, 1, &s);
6355 }
6356 expr instantiate(expr const &e, expr const &s) { return instantiate(e, 0, s); }
6357
6358 extern "C" object *lean_expr_instantiate1(object *a0, object *e0) {
6359     expr const &a = reinterpret_cast<expr const &>(a0);
6360     if (!has_loose_bvars(a)) {
6361         lean_inc(a0);
6362         return a0;
6363     }
6364     expr const &e = reinterpret_cast<expr const &>(e0);
6365     expr r = instantiate(a, 1, &e);
6366     return r.steal();
6367 }
6368
6369 static object *lean_expr_instantiate_core(b_obj_arg a0, size_t n,

```

```

6370                                     object **subst) {
6371     expr const &a = reinterpret_cast<expr const &>(a0);
6372     if (!has_loose_bvars(a) || n == 0) {
6373         lean_inc(a0);
6374         return a0;
6375     }
6376     expr r = replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
6377         if (offset >= get_loose_bvar_range(m))
6378             return some_expr(m); // expression m does not contain loose bound
6379                                     // variables with idx >= offset
6380         if (is_bvar(m)) {
6381             nat const &vidx = bvar_idx(m);
6382             if (vidx >= offset) {
6383                 size_t h = offset + n;
6384                 if (h < offset /* overflow, h is bigger than any vidx */ ||
6385                     (vidx.is_small() && vidx.get_small_value() < h)) {
6386                     object *v = subst[vidx.get_small_value() - offset];
6387                     return some_expr(lift_loose_bvars(TO_REF(expr, v), offset));
6388                 } else {
6389                     return some_expr(mk_bvar(vidx - nat::of_size_t(n)));
6390                 }
6391             }
6392         }
6393         return none_expr();
6394     });
6395     return r.steal();
6396 }
6397
6398 extern "C" object *lean_expr_instantiate(b_obj_arg a, b_obj_arg subst) {
6399     return lean_expr_instantiate_core(a, lean_array_size(subst),
6400                                     lean_array_cptr(subst));
6401 }
6402
6403 extern "C" object *lean_expr_instantiate_range(b_obj_arg a, b_obj_arg begin,
6404                                               b_obj_arg end, b_obj_arg subst) {
6405     if (!lean_is_scalar(begin) || !lean_is_scalar(end)) {
6406         lean_internal_panic("invalid range for Expr.instantiateRange");
6407     } else {
6408         usize sz = lean_array_size(subst);
6409         usize b = lean_unbox(begin);
6410         usize e = lean_unbox(end);
6411         if (b > e || e > sz) {
6412             lean_internal_panic("invalid range for Expr.instantiateRange");
6413         }
6414         return lean_expr_instantiate_core(a, e - b, lean_array_cptr(subst) + b);
6415     }
6416 }
6417
6418 expr instantiate_rev(expr const &a, unsigned n, expr const *subst) {
6419     if (!has_loose_bvars(a)) return a;
6420     return replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
6421         if (offset >= get_loose_bvar_range(m))
6422             return some_expr(m); // expression m does not contain loose bound
6423                                     // variables with idx >= offset
6424         if (is_bvar(m)) {
6425             nat const &vidx = bvar_idx(m);
6426             if (vidx >= offset) {
6427                 size_t h = offset + n;
6428                 if (h < offset /* overflow, h is bigger than any vidx */ ||
6429                     (vidx.is_small() && vidx.get_small_value() < h)) {
6430                     return some_expr(lift_loose_bvars(
6431                         subst[n - (vidx.get_small_value() - offset) - 1],
6432                         offset));
6433                 } else {
6434                     return some_expr(mk_bvar(vidx - nat(n)));
6435                 }
6436             }
6437         }
6438         return none_expr();
6439     });

```



```

6440 }
6441
6442 static object *lean_expr_instantiate_rev_core(object *a0, size_t n,
6443                                             object **subst) {
6444     expr const &a = reinterpret_cast<expr const &>(a0);
6445     if (!has_loose_bvars(a)) {
6446         lean_inc(a0);
6447         return a0;
6448     }
6449     expr r = replace(a, [=](expr const &m, unsigned offset) -> optional<expr> {
6450         if (offset >= get_loose_bvar_range(m))
6451             return some_expr(m); // expression m does not contain loose bound
6452                                 // variables with idx >= offset
6453         if (is_bvar(m)) {
6454             nat const &vidx = bvar_idx(m);
6455             if (vidx >= offset) {
6456                 size_t h = offset + n;
6457                 if (h < offset /* overflow, h is bigger than any vidx */ ||
6458                     (vidx.is_small() && vidx.get_small_value() < h)) {
6459                     object *v =
6460                         subst[n - (vidx.get_small_value() - offset) - 1];
6461                     return some_expr(lift_loose_bvars(TO_REF(expr, v), offset));
6462                 } else {
6463                     return some_expr(mk_bvar(vidx - nat::of_size_t(n)));
6464                 }
6465             }
6466         }
6467         return none_expr();
6468     });
6469     return r.steal();
6470 }
6471
6472 extern "C" object *lean_expr_instantiate_rev(b_obj_arg a, b_obj_arg subst) {
6473     return lean_expr_instantiate_rev_core(a, lean_array_size(subst),
6474                                         lean_array_cptr(subst));
6475 }
6476
6477 extern "C" object *lean_expr_instantiate_rev_range(b_obj_arg a, b_obj_arg begin,
6478                                                    b_obj_arg end,
6479                                                    b_obj_arg subst) {
6480     if (!lean_is_scalar(begin) || !lean_is_scalar(end)) {
6481         lean_internal_panic("invalid range for Expr.instantiateRevRange");
6482     } else {
6483         usize sz = lean_array_size(subst);
6484         usize b = lean_unbox(begin);
6485         usize e = lean_unbox(end);
6486         if (b > e || e > sz) {
6487             lean_internal_panic("invalid range for Expr.instantiateRevRange");
6488         }
6489         return lean_expr_instantiate_rev_core(a, e - b,
6490                                             lean_array_cptr(subst) + b);
6491     }
6492 }
6493
6494 bool is_head_beta(expr const &t) {
6495     return is_app(t) && is_lambda(get_app_fn(t));
6496 }
6497
6498 expr apply_beta(expr f, unsigned num_args, expr const *args) {
6499     if (num_args == 0) {
6500         return f;
6501     } else if (!is_lambda(f)) {
6502         return mk_rev_app(f, num_args, args);
6503     } else {
6504         unsigned m = 1;
6505         while (is_lambda(binding_body(f)) && m < num_args) {
6506             f = binding_body(f);
6507             m++;
6508         }
6509         lean_assert(m <= num_args);

```



```

6510         return mk_rev_app(
6511             instantiate(binding_body(f), m, args + (num_args - m)),
6512             num_args - m, args);
6513     }
6514 }
6515
6516 expr head_beta_reduce(expr const &t) {
6517     if (!is_head_beta(t)) {
6518         return t;
6519     } else {
6520         buffer<expr> args;
6521         expr const &f = get_app_rev_args(t, args);
6522         lean_assert(is_lambda(f));
6523         return head_beta_reduce(apply_beta(f, args.size(), args.data()));
6524     }
6525 }
6526
6527 expr cheap_beta_reduce(expr const &e) {
6528     if (!is_app(e)) return e;
6529     expr fn = get_app_fn(e);
6530     if (!is_lambda(fn)) return e;
6531     buffer<expr> args;
6532     get_app_args(e, args);
6533     unsigned i = 0;
6534     while (is_lambda(fn) && i < args.size()) {
6535         i++;
6536         fn = binding_body(fn);
6537     }
6538     if (!has_loose_bvars(fn)) {
6539         return mk_app(fn, args.size() - i, args.data() + i);
6540     } else if (is_bvar(fn)) {
6541         lean_assert(bvar_idx(fn) < i);
6542         return mk_app(args[i - bvar_idx(fn).get_small_value() - 1],
6543             args.size() - i, args.data() + i);
6544     } else {
6545         return e;
6546     }
6547 }
6548
6549 expr instantiate_lparams(expr const &e, names const &lps, levels const &ls) {
6550     if (!has_param_univ(e)) return e;
6551     return replace(e, [&](expr const &e) -> optional<expr> {
6552         if (!has_param_univ(e)) return some_expr(e);
6553         if (is_constant(e)) {
6554             return some_expr(update_constant(
6555                 e, map_reuse(const_levels(e), [&](level const &l) {
6556                     return instantiate(l, lps, ls);
6557                 })));
6558         } else if (is_sort(e)) {
6559             return some_expr(
6560                 update_sort(e, instantiate(sort_level(e), lps, ls)));
6561         } else {
6562             return none_expr();
6563         }
6564     });
6565 }
6566
6567 expr instantiate_type_lparams(constant_info const &info, levels const &ls) {
6568     if (info.get_num_lparams() != length(ls))
6569         lean_internal_panic(
6570             "#universes mismatch at instantiateTypeLevelParams");
6571     if (is_nil(ls) || !has_param_univ(info.get_type())) return info.get_type();
6572     return instantiate_lparams(info.get_type(), info.get_lparams(), ls);
6573 }
6574
6575 expr instantiate_value_lparams(constant_info const &info, levels const &ls) {
6576     if (info.get_num_lparams() != length(ls))
6577         lean_internal_panic(
6578             "#universes mismatch at instantiateValueLevelParams");
6579     if (!info.has_value())

```

```

6580     lean_internal_panic(
6581         "definition/theorem expected at instantiateValueLevelParams");
6582     if (is_nil(ls) || !has_param_univ(info.get_value()))
6583         return info.get_value();
6584     return instantiate_lparams(info.get_value(), info.get_lparams(), ls);
6585 }
6586
6587 extern "C" object *lean_instantiate_type_lparams(b_obj_arg info, b_obj_arg ls) {
6588     return instantiate_type_lparams(TO_REF(constant_info, info),
6589                                     TO_REF(levels, ls))
6590     .steal();
6591 }
6592
6593 extern "C" object *lean_instantiate_value_lparams(b_obj_arg info,
6594                                                  b_obj_arg ls) {
6595     return instantiate_value_lparams(TO_REF(constant_info, info),
6596                                     TO_REF(levels, ls))
6597     .steal();
6598 }
6599 } // namespace lean
6600 // ::::::::::::::
6601 // level.cpp
6602 // ::::::::::::::
6603 /*
6604 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
6605 Released under Apache 2.0 license as described in the file LICENSE.
6606
6607 Author: Leonardo de Moura
6608 */
6609 #include <lean/debug.h>
6610 #include <lean/hash.h>
6611 #include <lean/interrupt.h>
6612
6613 #include <algorithm>
6614 #include <unordered_set>
6615 #include <utility>
6616 #include <vector>
6617
6618 #include "kernel/environment.h"
6619 #include "kernel/level.h"
6620 #include "util/buffer.h"
6621 #include "util/list.h"
6622
6623 namespace lean {
6624
6625 extern "C" usize lean_level_hash(obj_arg l);
6626 extern "C" unsigned lean_level_depth(obj_arg l);
6627 extern "C" uint8 lean_level_has_mvar(obj_arg l);
6628 extern "C" uint8 lean_level_has_param(obj_arg l);
6629
6630 extern "C" object *lean_level_mk_zero(object *);
6631 extern "C" object *lean_level_mk_succ(obj_arg);
6632 extern "C" object *lean_level_mk_mvar(obj_arg);
6633 extern "C" object *lean_level_mk_param(obj_arg);
6634 extern "C" object *lean_level_mk_max(obj_arg, obj_arg);
6635 extern "C" object *lean_level_mk_imax(obj_arg, obj_arg);
6636 extern "C" object *lean_level_mk_max_simp(obj_arg, obj_arg);
6637 extern "C" object *lean_level_mk_imax_simp(obj_arg, obj_arg);
6638
6639 level mk_succ(level const &l) {
6640     return level(lean_level_mk_succ(l.to_obj_arg()));
6641 }
6642 level mk_max_core(level const &l1, level const &l2) {
6643     return level(lean_level_mk_max(l1.to_obj_arg(), l2.to_obj_arg()));
6644 }
6645 level mk_imax_core(level const &l1, level const &l2) {
6646     return level(lean_level_mk_imax(l1.to_obj_arg(), l2.to_obj_arg()));
6647 }
6648 level mk_univ_param(name const &n) {
6649     return level(lean_level_mk_param(n.to_obj_arg()));

```

```

6650 }
6651 level mk_univ_mvar(name const &n) {
6652     return level(lean_level_mk_mvar(n.to_obj_arg()));
6653 }
6654
6655 unsigned level::hash() const { return lean_level_hash(to_obj_arg()); }
6656 unsigned get_depth(level const &l) { return lean_level_depth(l.to_obj_arg()); }
6657 bool has_param(level const &l) { return lean_level_has_param(l.to_obj_arg()); }
6658 bool has_mvar(level const &l) { return lean_level_has_mvar(l.to_obj_arg()); }
6659
6660 bool is_explicit(level const &l) {
6661     switch (kind(l)) {
6662     case level_kind::Zero:
6663         return true;
6664     case level_kind::Param:
6665     case level_kind::MVar:
6666     case level_kind::Max:
6667     case level_kind::IMax:
6668         return false;
6669     case level_kind::Succ:
6670         return is_explicit(succ_of(l));
6671     }
6672     lean_unreachable(); // LCOV_EXCL_LINE
6673 }
6674
6675 /** \brief Convert (succ^k l) into (l, k). If l is not a succ, then return (l,
6676 * 0) */
6677 pair<level, unsigned> to_offset(level l) {
6678     unsigned k = 0;
6679     while (is_succ(l)) {
6680         l = succ_of(l);
6681         k++;
6682     }
6683     return mk_pair(l, k);
6684 }
6685
6686 unsigned to_explicit(level const &l) {
6687     lean_assert(is_explicit(l));
6688     return to_offset(l).second;
6689 }
6690
6691 level mk_max(level const &l1, level const &l2) {
6692     if (is_explicit(l1) && is_explicit(l2)) {
6693         return get_depth(l1) >= get_depth(l2) ? l1 : l2;
6694     } else if (l1 == l2) {
6695         return l1;
6696     } else if (is_zero(l1)) {
6697         return l2;
6698     } else if (is_zero(l2)) {
6699         return l1;
6700     } else if (is_max(l2) && (max_lhs(l2) == l1 || max_rhs(l2) == l1)) {
6701         return l2; // if l2 == (max l1 l'), then max l1 l2 == l2
6702     } else if (is_max(l1) && (max_lhs(l1) == l2 || max_rhs(l1) == l2)) {
6703         return l1; // if l1 == (max l2 l'), then max l1 l2 == l1
6704     } else {
6705         auto p1 = to_offset(l1);
6706         auto p2 = to_offset(l2);
6707         if (p1.first == p2.first) {
6708             lean_assert(p1.second != p2.second);
6709             return p1.second > p2.second ? l1 : l2;
6710         } else {
6711             return mk_max_core(l1, l2);
6712         }
6713     }
6714 }
6715
6716 level mk_imax(level const &l1, level const &l2) {
6717     if (is_not_zero(l2))
6718         return mk_max(l1, l2);
6719     else if (is_zero(l2))

```

```

6720     return l2; //  $\text{imax } u \ 0 = 0$  for any u
6721 else if (is_zero(l1))
6722     return l2; //  $\text{imax } 0 \ u = u$  for any u
6723 else if (l1 == l2)
6724     return l1; //  $\text{imax } u \ u = u$ 
6725 else
6726     return mk_imax_core(l1, l2);
6727 }
6728
6729 static level *g_level_zero = nullptr;
6730 static level *g_level_one = nullptr;
6731 level const &mk_level_zero() { return *g_level_zero; }
6732 level const &mk_level_one() { return *g_level_one; }
6733 bool is_one(level const &l) { return l == mk_level_one(); }
6734
6735 bool operator==(level const &l1, level const &l2) {
6736     if (kind(l1) != kind(l2)) return false;
6737     if (hash(l1) != hash(l2)) return false;
6738     if (is_eqp(l1, l2)) return true;
6739     switch (kind(l1)) {
6740     case level_kind::Zero:
6741         return true;
6742     case level_kind::Param:
6743     case level_kind::MVar:
6744         return level_id(l1) == level_id(l2);
6745     case level_kind::Max:
6746     case level_kind::IMax:
6747     case level_kind::Succ:
6748         if (get_depth(l1) != get_depth(l2)) return false;
6749         break;
6750     }
6751     switch (kind(l1)) {
6752     case level_kind::Zero:
6753     case level_kind::Param:
6754     case level_kind::MVar:
6755         lean_unreachable(); // LCOV_EXCL_LINE
6756     case level_kind::Max:
6757     case level_kind::IMax:
6758         return level_lhs(l1) == level_lhs(l2) &&
6759             level_rhs(l1) == level_rhs(l2);
6760     case level_kind::Succ:
6761         return succ_of(l1) == succ_of(l2);
6762     }
6763     lean_unreachable(); // LCOV_EXCL_LINE
6764 }
6765
6766 extern "C" uint8 lean_level_eqv(object *l1, object *l2) {
6767     return is_equivalent(TO_REF(level, l1), TO_REF(level, l2));
6768 }
6769
6770 extern "C" uint8 lean_level_eq(object *l1, object *l2) {
6771     return TO_REF(level, l1) == TO_REF(level, l2);
6772 }
6773
6774 bool is_not_zero(level const &l) {
6775     switch (kind(l)) {
6776     case level_kind::Zero:
6777     case level_kind::Param:
6778     case level_kind::MVar:
6779         return false;
6780     case level_kind::Succ:
6781         return true;
6782     case level_kind::Max:
6783         return is_not_zero(max_lhs(l)) || is_not_zero(max_rhs(l));
6784     case level_kind::IMax:
6785         return is_not_zero(imax_rhs(l));
6786     }
6787     lean_unreachable(); // LCOV_EXCL_LINE
6788 }
6789

```

```

6790 bool is_lt(level const &a, level const &b, bool use_hash) {
6791     if (is_eqp(a, b)) return false;
6792     unsigned da = get_depth(a);
6793     unsigned db = get_depth(b);
6794     if (da < db) return true;
6795     if (da > db) return false;
6796     if (kind(a) != kind(b)) return kind(a) < kind(b);
6797     if (use_hash) {
6798         if (hash(a) < hash(b)) return true;
6799         if (hash(a) > hash(b)) return false;
6800     }
6801     if (a == b) return false;
6802     switch (kind(a)) {
6803     case level_kind::Zero:
6804         lean_unreachable(); // LCOV_EXCL_LINE
6805     case level_kind::Param:
6806     case level_kind::MVar:
6807         return level_id(a) < level_id(b);
6808     case level_kind::Max:
6809     case level_kind::IMax:
6810         if (level_lhs(a) != level_lhs(b))
6811             return is_lt(level_lhs(a), level_lhs(b), use_hash);
6812         else
6813             return is_lt(level_rhs(a), level_rhs(b), use_hash);
6814     case level_kind::Succ:
6815         return is_lt(succ_of(a), succ_of(b), use_hash);
6816     }
6817     lean_unreachable(); // LCOV_EXCL_LINE
6818 }
6819
6820 bool is_lt(levels const &as, levels const &bs, bool use_hash) {
6821     if (is_nil(as)) return !is_nil(bs);
6822     if (is_nil(bs)) return false;
6823     if (car(as) == car(bs))
6824         return is_lt(cdr(as), cdr(bs), use_hash);
6825     else
6826         return is_lt(car(as), car(bs), use_hash);
6827 }
6828
6829 bool levels_has_param(b_obj_arg ls) {
6830     while (!is_scalar(ls)) {
6831         if (lean_level_has_param(cnstr_get(ls, 0))) return true;
6832         ls = cnstr_get(ls, 1);
6833     }
6834     return false;
6835 }
6836
6837 bool levels_has_mvar(b_obj_arg ls) {
6838     while (!is_scalar(ls)) {
6839         if (lean_level_has_mvar(cnstr_get(ls, 0))) return true;
6840         ls = cnstr_get(ls, 1);
6841     }
6842     return false;
6843 }
6844
6845 bool has_param(levels const &ls) { return levels_has_param(ls.raw()); }
6846 bool has_mvar(levels const &ls) { return levels_has_mvar(ls.raw()); }
6847
6848 void for_each_level_fn::apply(level const &l) {
6849     if (!m_f(l)) return;
6850     switch (l.kind()) {
6851     case level_kind::Succ:
6852         apply(succ_of(l));
6853         break;
6854     case level_kind::Max:
6855     case level_kind::IMax:
6856         apply(level_lhs(l));
6857         apply(level_rhs(l));
6858         break;
6859     case level_kind::Zero:

```

```

6860         case level_kind::Param:
6861         case level_kind::MVar:
6862             break;
6863     }
6864 }
6865
6866 level replace_level_fn::apply(level const &l) {
6867     optional<level> r = m_f(l);
6868     if (r) return *r;
6869     switch (l.kind()) {
6870         case level_kind::Succ:
6871             return update_succ(l, apply(succ_of(l)));
6872         case level_kind::Max:
6873         case level_kind::IMax: {
6874             level l1 = apply(level_lhs(l));
6875             level l2 = apply(level_rhs(l));
6876             return update_max(l, l1, l2);
6877         }
6878         case level_kind::Zero:
6879         case level_kind::Param:
6880         case level_kind::MVar:
6881             return l;
6882     }
6883     lean_unreachable(); // LCOV_EXCL_LINE
6884 }
6885
6886 bool occurs(level const &u, level const &l) {
6887     bool found = false;
6888     for_each(l, [&](level const &l) {
6889         if (found) return false;
6890         if (l == u) {
6891             found = true;
6892             return false;
6893         }
6894     });
6895     return found;
6896 }
6897
6898 optional<name> get_undef_param(level const &l, names const &ps) {
6899     optional<name> r;
6900     for_each(l, [&](level const &l) {
6901         if (!has_param(l) || r) return false;
6902         if (is_param(l) &&
6903             std::find(ps.begin(), ps.end(), param_id(l)) == ps.end())
6904             r = param_id(l);
6905     });
6906     return r;
6907 }
6908
6909 level update_succ(level const &l, level const &new_arg) {
6910     if (is_eqp(succ_of(l), new_arg))
6911         return l;
6912     else
6913         return mk_succ(new_arg);
6914 }
6915
6916 level update_max(level const &l, level const &new_lhs, level const &new_rhs) {
6917     if (is_eqp(level_lhs(l), new_lhs) && is_eqp(level_rhs(l), new_rhs))
6918         return l;
6919     else if (is_max(l))
6920         return mk_max(new_lhs, new_rhs);
6921     else
6922         return mk_imax(new_lhs, new_rhs);
6923 }
6924
6925 extern "C" object *lean_level_update_succ(obj_arg l, obj_arg new_arg) {
6926     if (succ_of(TO_REF(level, l)).raw() == new_arg) {
6927         lean_dec(new_arg);

```

```

6930     return l;
6931 } else {
6932     lean_dec_ref(l);
6933     return lean_level_mk_succ(new_arg);
6934 }
6935 }
6936
6937 extern "C" object *lean_level_update_max(obj_arg l, obj_arg new_lhs,
6938                                         obj_arg new_rhs) {
6939     if (max_lhs(TO_REF(level, l)).raw() == new_lhs &&
6940         max_rhs(TO_REF(level, l)).raw() == new_rhs) {
6941         lean_dec(new_lhs);
6942         lean_dec(new_rhs);
6943         return l;
6944     } else {
6945         lean_dec_ref(l);
6946         return lean_level_mk_max_simp(new_lhs, new_rhs);
6947     }
6948 }
6949
6950 extern "C" object *lean_level_update_imax(obj_arg l, obj_arg new_lhs,
6951                                           obj_arg new_rhs) {
6952     if (imax_lhs(TO_REF(level, l)).raw() == new_lhs &&
6953         imax_rhs(TO_REF(level, l)).raw() == new_rhs) {
6954         lean_dec(new_lhs);
6955         lean_dec(new_rhs);
6956         return l;
6957     } else {
6958         lean_dec_ref(l);
6959         return lean_level_mk_imax_simp(new_lhs, new_rhs);
6960     }
6961 }
6962
6963 level instantiate(level const &l, names const &ps, levels const &ls) {
6964     lean_assert(length(ps) == length(ls));
6965     return replace(l, [=](level const &l) {
6966         if (!has_param(l)) {
6967             return some_level(l);
6968         } else if (is_param(l)) {
6969             name const &id = param_id(l);
6970             names const *it1 = &ps;
6971             levels const *it2 = &ls;
6972             /* The assertion above ensures that !is_nil(*it2) is unnecessary, but
6973              we we keep it here to ensure the lean_instantiate_lparams does
6974              not crash at runtime when misused. */
6975             while (!is_nil(*it1) && !is_nil(*it2)) {
6976                 if (head(*it1) == id) return some_level(head(*it2));
6977                 it1 = &tail(*it1);
6978                 it2 = &tail(*it2);
6979             }
6980             return some_level(l);
6981         } else {
6982             return none_level();
6983         }
6984     });
6985 }
6986
6987 static void print(std::ostream &out, level l);
6988
6989 static void print_child(std::ostream &out, level const &l) {
6990     if (is_explicit(l) || is_param(l) || is_mvar(l)) {
6991         print(out, l);
6992     } else {
6993         out << "(";
6994         print(out, l);
6995         out << ")";
6996     }
6997 }
6998
6999 static void print(std::ostream &out, level l) {

```



```

7000     if (is_explicit(l)) {
7001         out << get_depth(l);
7002     } else {
7003         switch (kind(l)) {
7004             case level_kind::Zero:
7005                 lean_unreachable(); // LCOV_EXCL_LINE
7006             case level_kind::Param:
7007                 out << param_id(l);
7008                 break;
7009             case level_kind::MVar:
7010                 out << "?" << mvar_id(l);
7011                 break;
7012             case level_kind::Succ:
7013                 out << "succ ";
7014                 print_child(out, succ_of(l));
7015                 break;
7016             case level_kind::Max:
7017             case level_kind::IMax:
7018                 if (is_max(l))
7019                     out << "max ";
7020                 else
7021                     out << "imax ";
7022                 print_child(out, level_lhs(l));
7023                 // max and imax are right associative
7024                 while (kind(level_rhs(l)) == kind(l)) {
7025                     l = level_rhs(l);
7026                     out << " ";
7027                     print_child(out, level_lhs(l));
7028                 }
7029                 out << " ";
7030                 print_child(out, level_rhs(l));
7031                 break;
7032         }
7033     }
7034 }
7035
7036 std::ostream &operator<<(std::ostream &out, level const &l) {
7037     print(out, l);
7038     return out;
7039 }
7040
7041 format pp(level l, bool unicode, unsigned indent);
7042
7043 static format pp_child(level const &l, bool unicode, unsigned indent) {
7044     if (is_explicit(l) || is_param(l) || is_mvar(l)) {
7045         return pp(l, unicode, indent);
7046     } else {
7047         return paren(pp(l, unicode, indent));
7048     }
7049 }
7050
7051 format pp(level l, bool unicode, unsigned indent) {
7052     if (is_explicit(l)) {
7053         return format(get_depth(l));
7054     } else {
7055         switch (kind(l)) {
7056             case level_kind::Zero:
7057                 lean_unreachable(); // LCOV_EXCL_LINE
7058             case level_kind::Param:
7059                 return format(param_id(l));
7060             case level_kind::MVar:
7061                 return format("?") + format(mvar_id(l));
7062             case level_kind::Succ: {
7063                 auto p = to_offset(l);
7064                 auto fmt1 = pp_child(p.first, unicode, indent);
7065                 return fmt1 + format("+") + format(p.second);
7066             }
7067             case level_kind::Max:
7068             case level_kind::IMax: {
7069                 format r = format(is_max(l) ? "max" : "imax");

```

```

7070         r += nest(indent, compose(line(), pp_child(level_lhs(l),
7071                                                     unicode, indent)));
7072         // max and imax are right associative
7073         while (kind(level_rhs(l)) == kind(l)) {
7074             l = level_rhs(l);
7075             r += nest(indent,
7076                     compose(line(),
7077                             pp_child(level_lhs(l), unicode, indent)));
7078         }
7079         r += nest(indent, compose(line(), pp_child(level_rhs(l),
7080                                                     unicode, indent)));
7081         return group(r);
7082     }
7083 }
7084 lean_unreachable(); // LCOV_EXCL_LINE
7085 }
7086 }
7087
7088 format pp(level const &l, options const &opts) {
7089     return pp(l, get_pp_unicode(opts), get_pp_indent(opts));
7090 }
7091
7092 format pp(level const &lhs, level const &rhs, bool unicode, unsigned indent) {
7093     format leq = unicode ? format("<=") : format("<");
7094     return group(pp(lhs, unicode, indent) + space() + leq + line() +
7095                pp(rhs, unicode, indent));
7096 }
7097
7098 format pp(level const &lhs, level const &rhs, options const &opts) {
7099     return pp(lhs, rhs, get_pp_unicode(opts), get_pp_indent(opts));
7100 }
7101
7102 // A total order on level expressions that has the following properties
7103 // - succ(l) is an immediate successor of l.
7104 // - zero is the minimal element.
7105 // This total order is used in the normalization procedure.
7106 static bool is_norm_lt(level const &a, level const &b) {
7107     if (is_eqp(a, b)) return false;
7108     auto p1 = to_offset(a);
7109     auto p2 = to_offset(b);
7110     level const &l1 = p1.first;
7111     level const &l2 = p2.first;
7112     if (l1 != l2) {
7113         if (kind(l1) != kind(l2)) return kind(l1) < kind(l2);
7114         switch (kind(l1)) {
7115             case level_kind::Zero:
7116             case level_kind::Succ:
7117                 lean_unreachable(); // LCOV_EXCL_LINE
7118             case level_kind::Param:
7119             case level_kind::MVar:
7120                 return level_id(l1) < level_id(l2);
7121             case level_kind::Max:
7122             case level_kind::IMax:
7123                 if (level_lhs(l1) != level_lhs(l2))
7124                     return is_norm_lt(level_lhs(l1), level_lhs(l2));
7125                 else
7126                     return is_norm_lt(level_rhs(l1), level_rhs(l2));
7127         }
7128         lean_unreachable(); // LCOV_EXCL_LINE
7129     } else {
7130         return p1.second < p2.second;
7131     }
7132 }
7133
7134 void push_max_args(level const &l, buffer<level> &r) {
7135     if (is_max(l)) {
7136         push_max_args(max_lhs(l), r);
7137         push_max_args(max_rhs(l), r);
7138     } else {
7139         r.push_back(l);

```

```

7140     }
7141 }
7142
7143 level mk_max(buffer<level> const &args) {
7144     lean_assert(!args.empty());
7145     unsigned nargs = args.size();
7146     if (nargs == 1) {
7147         return args[0];
7148     } else {
7149         lean_assert(nargs >= 2);
7150         level r = mk_max(args[nargs - 2], args[nargs - 1]);
7151         unsigned i = nargs - 2;
7152         while (i > 0) {
7153             --i;
7154             r = mk_max(args[i], r);
7155         }
7156         return r;
7157     }
7158 }
7159
7160 level mk_succ(level l, unsigned k) {
7161     while (k > 0) {
7162         --k;
7163         l = mk_succ(l);
7164     }
7165     return l;
7166 }
7167
7168 level normalize(level const &l) {
7169     auto p = to_offset(l);
7170     level const &r = p.first;
7171     switch (kind(r)) {
7172     case level_kind::Succ:
7173         lean_unreachable(); // LCOV_EXCL_LINE
7174     case level_kind::Zero:
7175     case level_kind::Param:
7176     case level_kind::MVar:
7177         return l;
7178     case level_kind::IMax: {
7179         auto l1 = normalize(imax_lhs(r));
7180         auto l2 = normalize(imax_rhs(r));
7181         return mk_imax(l1, l2);
7182     }
7183     case level_kind::Max: {
7184         buffer<level> todo;
7185         buffer<level> args;
7186         push_max_args(r, todo);
7187         for (level const &a : todo) push_max_args(normalize(a), args);
7188         std::sort(args.begin(), args.end(), is_norm_lt);
7189         buffer<level> &rargs = todo;
7190         rargs.clear();
7191         unsigned i = 0;
7192         if (is_explicit(args[i])) {
7193             // find max explicit universe
7194             while (i + 1 < args.size() && is_explicit(args[i + 1])) i++;
7195             lean_assert(is_explicit(args[i]));
7196             unsigned k = to_offset(args[i]).second;
7197             // an explicit universe k is subsumed by succ^k(l)
7198             unsigned j = i + 1;
7199             for (; j < args.size(); j++) {
7200                 if (to_offset(args[j]).second >= k) break;
7201             }
7202             if (j < args.size()) {
7203                 // explicit universe was subsumed by succ^k'(l) where k' >=
7204                 // k
7205                 i++;
7206             }
7207         }
7208         rargs.push_back(args[i]);
7209         auto p_prev = to_offset(args[i]);

```

```

7210         i++;
7211         for (; i < args.size(); i++) {
7212             auto p_curr = to_offset(args[i]);
7213             if (p_prev.first == p_curr.first) {
7214                 if (p_prev.second < p_curr.second) {
7215                     p_prev = p_curr;
7216                     rargs.pop_back();
7217                     rargs.push_back(args[i]);
7218                 }
7219             } else {
7220                 p_prev = p_curr;
7221                 rargs.push_back(args[i]);
7222             }
7223         }
7224         for (level &a : rargs) a = mk_succ(a, p.second);
7225         return mk_max(rargs);
7226     }
7227 }
7228 lean_unreachable(); // LCOV_EXCL_LINE
7229 }
7230
7231 bool is_equivalent(level const &lhs, level const &rhs) {
7232     check_system("level constraints");
7233     return lhs == rhs || normalize(lhs) == normalize(rhs);
7234 }
7235
7236 bool is_geq_core(level l1, level l2) {
7237     if (l1 == l2 || is_zero(l2)) return true;
7238     if (is_max(l2)) return is_geq(l1, max_lhs(l2)) && is_geq(l1, max_rhs(l2));
7239     if (is_max(l1) && (is_geq(max_lhs(l1), l2) || is_geq(max_rhs(l1), l2)))
7240         return true;
7241     if (is_imax(l2))
7242         return is_geq(l1, imax_lhs(l2)) && is_geq(l1, imax_rhs(l2));
7243     if (is_imax(l1)) return is_geq(imax_rhs(l1), l2);
7244     auto p1 = to_offset(l1);
7245     auto p2 = to_offset(l2);
7246     if (p1.first == p2.first || is_zero(p2.first))
7247         return p1.second >= p2.second;
7248     if (p1.second == p2.second && p1.second > 0)
7249         return is_geq(p1.first, p2.first);
7250     return false;
7251 }
7252 bool is_geq(level const &l1, level const &l2) {
7253     return is_geq_core(normalize(l1), normalize(l2));
7254 }
7255 levels lparams_to_levels(names const &ps) {
7256     buffer<level> ls;
7257     for (auto const &p : ps) ls.push_back(mk_univ_param(p));
7258     return levels(ls);
7259 }
7260
7261 level::level() : level(*g_level_zero) {}
7262
7263 void initialize_level() {
7264     g_level_zero = new level(lean_level_mk_zero(box(0)));
7265     mark_persistent(g_level_zero->raw());
7266     g_level_one = new level(mk_succ(*g_level_zero));
7267     mark_persistent(g_level_one->raw());
7268 }
7269
7270 void finalize_level() {
7271     delete g_level_one;
7272     delete g_level_zero;
7273 }
7274 } // namespace lean
7275 void print(lean::level const &l) { std::cout << l << std::endl; }
7276 // ::::::::::::::
7277 // local_ctx.cpp
7278 // ::::::::::::::
7279 /*

```

```

7280 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
7281 Released under Apache 2.0 license as described in the file LICENSE.
7282
7283 Author: Leonardo de Moura
7284 */
7285 #include <lean/sstream.h>
7286
7287 #include <limits>
7288
7289 #include "kernel/abstract.h"
7290 #include "kernel/local_ctx.h"
7291
7292 namespace lean {
7293 static expr *g_dummy_type;
7294 static local_decl *g_dummy_decl;
7295
7296 extern "C" object *lean_mk_local_decl(object *index, object *fvarid,
7297                                     object *user_name, object *type,
7298                                     uint8 bi);
7299 extern "C" object *lean_mk_let_decl(object *index, object *fvarid,
7300                                   object *user_name, object *type,
7301                                   object *val);
7302 extern "C" uint8 lean_local_decl_binder_info(object *d);
7303
7304 local_decl::local_decl() : object_ref(*g_dummy_decl) {}
7305
7306 local_decl::local_decl(unsigned idx, name const &n, name const &un,
7307                        expr const &t, expr const &v)
7308     : object_ref(lean_mk_let_decl(nat(idx).to_obj_arg(), n.to_obj_arg(),
7309                                  un.to_obj_arg(), t.to_obj_arg(),
7310                                  v.to_obj_arg())) {}
7311
7312 local_decl::local_decl(unsigned idx, name const &n, name const &un,
7313                        expr const &t, binder_info bi)
7314     : object_ref(lean_mk_local_decl(nat(idx).to_obj_arg(), n.to_obj_arg(),
7315                                    un.to_obj_arg(), t.to_obj_arg(),
7316                                    static_cast<uint8>(bi))) {}
7317
7318 local_decl::local_decl(local_decl const &d, expr const &t, expr const &v)
7319     : local_decl(d.get_idx(), d.get_name(), d.get_user_name(), t, v) {}
7320
7321 local_decl::local_decl(local_decl const &d, expr const &t)
7322     : local_decl(d.get_idx(), d.get_name(), d.get_user_name(), t,
7323                 d.get_info()) {}
7324
7325 binder_info local_decl::get_info() const {
7326     return static_cast<binder_info>(lean_local_decl_binder_info(to_obj_arg()));
7327 }
7328
7329 expr local_decl::mk_ref() const { return mk_fvar(get_name()); }
7330
7331 extern "C" object *lean_mk_empty_local_ctx(object *);
7332 extern "C" object *lean_local_ctx_num_indices(object *);
7333 extern "C" uint8 lean_local_ctx_is_empty(object *);
7334 extern "C" object *lean_local_ctx_mk_local_decl(object *lctx, object *name,
7335                                                  object *user_name, object *expr,
7336                                                  uint8 bi);
7337 extern "C" object *lean_local_ctx_mk_let_decl(object *lctx, object *name,
7338                                               object *user_name, object *type,
7339                                               object *value);
7340 extern "C" object *lean_local_ctx_find(object *lctx, object *name);
7341 extern "C" object *lean_local_ctx_erase(object *lctx, object *name);
7342
7343 local_ctx::local_ctx() : object_ref(lean_mk_empty_local_ctx(box(0))) {}
7344
7345 bool local_ctx::empty() const { return lean_local_ctx_is_empty(to_obj_arg()); }
7346
7347 local_decl local_ctx::mk_local_decl(name const &n, name const &un,
7348                                   expr const &type, expr const &value) {
7349     unsigned idx = unbox(lean_local_ctx_num_indices(to_obj_arg()));

```

```

7350     m_obj = lean_local_ctx_mk_let_decl(raw(), n.to_obj_arg(), un.to_obj_arg(),
7351                                     type.to_obj_arg(), value.to_obj_arg());
7352     return local_decl(idx, n, un, type, value);
7353 }
7354
7355 local_decl local_ctx::mk_local_decl(name const &n, name const &un,
7356                                   expr const &type, binder_info bi) {
7357     unsigned idx = unbox(lean_local_ctx_num_indices(to_obj_arg()));
7358     m_obj =
7359         lean_local_ctx_mk_local_decl(raw(), n.to_obj_arg(), un.to_obj_arg(),
7360                                     type.to_obj_arg(), static_cast<uint8>(bi));
7361     return local_decl(idx, n, un, type, bi);
7362 }
7363
7364 optional<local_decl> local_ctx::find_local_decl(name const &n) const {
7365     return to_optional<local_decl>{
7366         lean_local_ctx_find(to_obj_arg(), n.to_obj_arg())};
7367 }
7368
7369 local_decl local_ctx::get_local_decl(name const &n) const {
7370     if (optional<local_decl> r = find_local_decl(n)) {
7371         return *r;
7372     } else {
7373         // lean_assert(false);
7374         throw exception(sstream() << "unknown free variable: " << n);
7375     }
7376 }
7377
7378 expr local_ctx::get_local(name const &n) const {
7379     lean_assert(find_local_decl(n));
7380     return get_local_decl(n).mk_ref();
7381 }
7382
7383 void local_ctx::clear(local_decl const &d) {
7384     m_obj = lean_local_ctx_erase(m_obj, d.get_name().to_obj_arg());
7385 }
7386
7387 template <bool is_lambda>
7388 expr local_ctx::mk_binding(unsigned num, expr const *fvars, expr const &b,
7389                           bool remove_dead_let) const {
7390     expr r = abstract(b, num, fvars);
7391     unsigned i = num;
7392     while (i > 0) {
7393         --i;
7394         local_decl const &decl = get_local_decl(fvar_name(fvars[i]));
7395         if (optional<expr> const &opt_val = decl.get_value()) {
7396             if (!remove_dead_let || has_loose_bvar(r, 0)) {
7397                 expr type = abstract(decl.get_type(), i, fvars);
7398                 expr value = abstract(*opt_val, i, fvars);
7399                 r = ::lean::mk_let(decl.get_user_name(), type, value, r);
7400             } else {
7401                 r = lower_loose_bvars(r, 1, 1);
7402             }
7403         } else if (is_lambda) {
7404             expr type = abstract(decl.get_type(), i, fvars);
7405             r = ::lean::mk_lambda(decl.get_user_name(), type, r,
7406                                 decl.get_info());
7407         } else {
7408             expr type = abstract(decl.get_type(), i, fvars);
7409             r = ::lean::mk_pi(decl.get_user_name(), type, r, decl.get_info());
7410         }
7411     }
7412     return r;
7413 }
7414
7415 expr local_ctx::mk_lambda(unsigned num, expr const *fvars, expr const &e,
7416                          bool remove_dead_let) const {
7417     return mk_binding<true>(num, fvars, e, remove_dead_let);
7418 }
7419

```

```

7420 expr local_ctx::mk_pi(unsigned num, expr const *fvars, expr const &e,
7421                        bool remove_dead_let) const {
7422     return mk_binding<false>(num, fvars, e, remove_dead_let);
7423 }
7424
7425 void initialize_local_ctx() {
7426     g_dummy_type = new expr(mk_constant(name::mk_internal_unique_name()));
7427     mark_persistent(g_dummy_type->raw());
7428     g_dummy_decl = new local_decl(std::numeric_limits<unsigned>::max(),
7429                                   name("__local_decl_for_default_constructor"),
7430                                   name("__local_decl_for_default_constructor"),
7431                                   mk_Prop(), mk_binder_info());
7432     mark_persistent(g_dummy_decl->raw());
7433 }
7434
7435 void finalize_local_ctx() {
7436     delete g_dummy_decl;
7437     delete g_dummy_type;
7438 }
7439 } // namespace lean
7440 // ::::::::::::::
7441 // quot.cpp
7442 // ::::::::::::::
7443 /*
7444 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
7445 Released under Apache 2.0 license as described in the file LICENSE.
7446
7447 Author: Leonardo de Moura
7448
7449 Quotient types.
7450 */
7451 #include "kernel/local_ctx.h"
7452 #include "kernel/quot.h"
7453 #include "util/name_generator.h"
7454
7455 namespace lean {
7456 name *quot_consts::g_quot = nullptr;
7457 name *quot_consts::g_quot_lift = nullptr;
7458 name *quot_consts::g_quot_ind = nullptr;
7459 name *quot_consts::g_quot_mk = nullptr;
7460
7461 static void check_eq_type(environment const &env) {
7462     constant_info eq_info = env.get("Eq");
7463     if (!eq_info.is_inductive())
7464         throw exception(
7465             "failed to initialize quot module, environment does not have 'Eq' "
7466             "type");
7467     inductive_val eq_val = eq_info.to_inductive_val();
7468     if (length(eq_info.get_lparams()) != 1)
7469         throw exception(
7470             "failed to initialize quot module, unexpected number of universe "
7471             "params at 'Eq' type");
7472     if (length(eq_val.get_cnstrs()) != 1)
7473         throw exception(
7474             "failed to initialize quot module, unexpected number of "
7475             "constructors for 'Eq' type");
7476     local_ctx lctx;
7477     name_generator g;
7478     {
7479         level u = mk_univ_param(head(eq_info.get_lparams()));
7480         expr alpha =
7481             lctx.mk_local_decl(g, "α", mk_sort(u), mk_implicit_binder_info());
7482         expr expected_eq_type =
7483             lctx.mk_pi(alpha, mk_arrow(alpha, mk_arrow(alpha, mk_Prop())));
7484         if (expected_eq_type != eq_info.get_type())
7485             throw exception(
7486                 "failed to initialize quot module, 'Eq' has an expected type");
7487     }
7488     {
7489         constant_info eq_refl_info = env.get(head(eq_val.get_cnstrs()));

```



```

7490     level u = mk_univ_param(head(eq_refl_info.get_lparams()));
7491     expr alpha =
7492         lctx.mk_local_decl(g, "α", mk_sort(u), mk_implicit_binder_info());
7493     expr a = lctx.mk_local_decl(g, "a", alpha);
7494     expr expected_eq_refl_type =
7495         lctx.mk_pi({alpha, a}, mk_app(mk_constant("Eq", {u}), alpha, a, a));
7496     if (eq_refl_info.get_type() != expected_eq_refl_type)
7497         throw exception(
7498             "failed to initialize quot module, unexpected type for 'Eq' "
7499             "type constructor");
7500 }
7501 }
7502
7503 environment environment::add_quot() const {
7504     if (is_quot_initialized()) return *this;
7505     check_eq_type(*this);
7506     environment new_env = *this;
7507     name u_name("u");
7508     local_ctx lctx;
7509     name_generator g;
7510     level u = mk_univ_param(u_name);
7511     expr Sort_u = mk_sort(u);
7512     expr alpha = lctx.mk_local_decl(g, "α", Sort_u, mk_implicit_binder_info());
7513     expr r =
7514         lctx.mk_local_decl(g, "r", mk_arrow(alpha, mk_arrow(alpha, mk_Prop())));
7515     /* constant {u} quot {α : Sort u} (r : α → α → Prop) : Sort u */
7516     new_env.add_core(constant_info(quot_val(*quot_consts::g_quot, {u_name},
7517         lctx.mk_pi({alpha, r}, Sort_u),
7518         quot_kind::Type)));
7519     expr quot_r = mk_app(mk_constant(*quot_consts::g_quot, {u}), alpha, r);
7520     expr a = lctx.mk_local_decl(g, "a", alpha);
7521     /* constant {u} quot.mk {α : Sort u} (r : α → α → Prop) (a : α) : @quot.{u}
7522        * α r */
7523     new_env.add_core(constant_info(quot_val(*quot_consts::g_quot_mk, {u_name},
7524         lctx.mk_pi({alpha, r, a}, quot_r),
7525         quot_kind::Mk)));
7526     /* make r implicit */
7527     lctx = local_ctx();
7528     alpha = lctx.mk_local_decl(g, "α", Sort_u, mk_implicit_binder_info());
7529     r = lctx.mk_local_decl(g, "r", mk_arrow(alpha, mk_arrow(alpha, mk_Prop())),
7530         mk_implicit_binder_info());
7531     quot_r = mk_app(mk_constant(*quot_consts::g_quot, {u}), alpha, r);
7532     a = lctx.mk_local_decl(g, "a", alpha);
7533     name v_name("v");
7534     level v = mk_univ_param(v_name);
7535     expr Sort_v = mk_sort(v);
7536     expr beta = lctx.mk_local_decl(g, "β", Sort_v, mk_implicit_binder_info());
7537     expr f = lctx.mk_local_decl(g, "f", mk_arrow(alpha, beta));
7538     expr b = lctx.mk_local_decl(g, "b", alpha);
7539     expr r_a_b = mk_app(r, a, b);
7540     /* f a = f b */
7541     expr f_a_eq_f_b =
7542         mk_app(mk_constant("Eq", {v}), beta, mk_app(f, a), mk_app(f, b));
7543     /* (∀ a b : α, r a b → f a = f b) */
7544     expr sanity = lctx.mk_pi({a, b}, mk_arrow(r_a_b, f_a_eq_f_b));
7545     /* constant {u v} quot.lift {α : Sort u} {r : α → α → Prop} {β : Sort v} (f
7546        : α → β) : (∀ a b : α, r a b → f a = f b) → @quot.{u} α r → β */
7547     new_env.add_core(constant_info(
7548         quot_val(*quot_consts::g_quot_lift, {u_name, v_name},
7549             lctx.mk_pi({alpha, r, beta, f},
7550                 mk_arrow(sanity, mk_arrow(quot_r, beta))),
7551         quot_kind::Lift)));
7552     /* {β : @quot.{u} α r → Prop} */
7553     beta = lctx.mk_local_decl(g, "β", mk_arrow(quot_r, mk_Prop()),
7554         mk_implicit_binder_info());
7555     expr quot_mk_a =
7556         mk_app(mk_constant(*quot_consts::g_quot_mk, {u}), alpha, r, a);
7557     expr all_quot = lctx.mk_pi(a, mk_app(beta, quot_mk_a));
7558     expr q = lctx.mk_local_decl(g, "q", quot_r);
7559     expr beta_q = mk_app(beta, q);

```

```

7560 /* constant {u} quot.ind {α : Sort u} {r : α → α → Prop} {β : @quot.{u} α r
7561      → Prop} : (∀ a : α, β (@quot.mk.{u} α r a)) → ∀ q : @quot.{u} α r, β q */
7562 new_env.add_core(constant_info(quot_val(
7563     *quot_consts::g_quot_ind, {u_name},
7564     lctx.mk_pi({alpha, r, beta}, mk_arrow(all_quot, lctx.mk_pi(q, beta_q))),
7565     quot_kind::Ind));
7566 new_env.mark_quot_initialized();
7567 return new_env;
7568 }
7569
7570 void initialize_quot() {
7571     quot_consts::g_quot = new name{"Quot"};
7572     mark_persistent(quot_consts::g_quot->raw());
7573     quot_consts::g_quot_lift = new name{"Quot", "lift"};
7574     mark_persistent(quot_consts::g_quot_lift->raw());
7575     quot_consts::g_quot_ind = new name{"Quot", "ind"};
7576     mark_persistent(quot_consts::g_quot_ind->raw());
7577     quot_consts::g_quot_mk = new name{"Quot", "mk"};
7578     mark_persistent(quot_consts::g_quot_mk->raw());
7579 }
7580
7581 void finalize_quot() {
7582     delete quot_consts::g_quot;
7583     delete quot_consts::g_quot_lift;
7584     delete quot_consts::g_quot_ind;
7585     delete quot_consts::g_quot_mk;
7586 }
7587 } // namespace lean
7588 // ::::::::::::::
7589 // replace_fn.cpp
7590 // ::::::::::::::
7591 /*
7592 Copyright (c) 2013-2014 Microsoft Corporation. All rights reserved.
7593 Released under Apache 2.0 license as described in the file LICENSE.
7594
7595 Author: Leonardo de Moura
7596 */
7597 #include <memory>
7598 #include <vector>
7599
7600 #include "kernel/cache_stack.h"
7601 #include "kernel/replace_fn.h"
7602
7603 #ifndef LEAN_DEFAULT_REPLACE_CACHE_CAPACITY
7604 #define LEAN_DEFAULT_REPLACE_CACHE_CAPACITY 1024 * 8
7605 #endif
7606
7607 namespace lean {
7608 struct replace_cache {
7609     struct entry {
7610         object *m_cell;
7611         unsigned m_offset;
7612         expr m_result;
7613         entry() : m_cell(nullptr) {}
7614     };
7615     unsigned m_capacity;
7616     std::vector<entry> m_cache;
7617     std::vector<unsigned> m_used;
7618     replace_cache(unsigned c) : m_capacity(c), m_cache(c) {}
7619
7620     expr *find(expr const &e, unsigned offset) {
7621         unsigned i = hash(hash(e), offset) % m_capacity;
7622         if (m_cache[i].m_cell == e.raw() && m_cache[i].m_offset == offset)
7623             return &m_cache[i].m_result;
7624         else
7625             return nullptr;
7626     }
7627
7628     void insert(expr const &e, unsigned offset, expr const &v) {
7629         unsigned i = hash(hash(e), offset) % m_capacity;

```

```

7630     if (m_cache[i].m_cell == nullptr) m_used.push_back(i);
7631     m_cache[i].m_cell = e.raw();
7632     m_cache[i].m_offset = offset;
7633     m_cache[i].m_result = v;
7634 }
7635
7636 void clear() {
7637     for (unsigned i : m_used) {
7638         m_cache[i].m_cell = nullptr;
7639         m_cache[i].m_result = expr();
7640     }
7641     m_used.clear();
7642 }
7643 };
7644
7645 /* CACHE_RESET: NO */
7646 MK_CACHE_STACK(replace_cache, LEAN_DEFAULT_REPLACE_CACHE_CAPACITY)
7647
7648 class replace_rec_fn {
7649     replace_cache_ref m_cache;
7650     std::function<optional<expr>(expr const &, unsigned)> m_f;
7651     bool m_use_cache;
7652
7653     expr save_result(expr const &e, unsigned offset, expr const &r,
7654                     bool shared) {
7655         if (shared) m_cache->insert(e, offset, r);
7656         return r;
7657     }
7658
7659     expr apply(expr const &e, unsigned offset) {
7660         bool shared = false;
7661         if (m_use_cache && is_shared(e)) {
7662             if (auto r = m_cache->find(e, offset)) return *r;
7663             shared = true;
7664         }
7665         check_system("replace");
7666
7667         if (optional<expr> r = m_f(e, offset)) {
7668             return save_result(e, offset, *r, shared);
7669         } else {
7670             switch (e.kind()) {
7671                 case expr_kind::Const:
7672                 case expr_kind::Sort:
7673                 case expr_kind::BVar:
7674                 case expr_kind::Lit:
7675                 case expr_kind::MVar:
7676                 case expr_kind::FVar:
7677                     return save_result(e, offset, e, shared);
7678                 case expr_kind::MData: {
7679                     expr new_e = apply(mdata_expr(e), offset);
7680                     return save_result(e, offset, update_mdata(e, new_e),
7681                                       shared);
7682                 }
7683                 case expr_kind::Proj: {
7684                     expr new_e = apply(proj_expr(e), offset);
7685                     return save_result(e, offset, update_proj(e, new_e),
7686                                       shared);
7687                 }
7688                 case expr_kind::App: {
7689                     expr new_f = apply(app_fn(e), offset);
7690                     expr new_a = apply(app_arg(e), offset);
7691                     return save_result(e, offset, update_app(e, new_f, new_a),
7692                                       shared);
7693                 }
7694                 case expr_kind::Pi:
7695                 case expr_kind::Lambda: {
7696                     expr new_d = apply(binding_domain(e), offset);
7697                     expr new_b = apply(binding_body(e), offset + 1);
7698                     return save_result(e, offset,
7699                                       update_binding(e, new_d, new_b), shared);
7700                 }

```

```

7700     }
7701     case expr_kind::Let: {
7702         expr new_t = apply(let_type(e), offset);
7703         expr new_v = apply(let_value(e), offset);
7704         expr new_b = apply(let_body(e), offset + 1);
7705         return save_result(
7706             e, offset, update_let(e, new_t, new_v, new_b), shared);
7707     }
7708 }
7709 lean_unreachable();
7710 }
7711 }
7712
7713 public:
7714     template <typename F>
7715     replace_rec_fn(F const &f, bool use_cache)
7716         : m_f(f), m_use_cache(use_cache) {}
7717
7718     expr operator()(expr const &e) { return apply(e, 0); }
7719 };
7720
7721 expr replace(expr const &e,
7722             std::function<optional<expr>(expr const &, unsigned)> const &f,
7723             bool use_cache) {
7724     return replace_rec_fn(f, use_cache)(e);
7725 }
7726 } // namespace lean
7727 // ::::::::::::::
7728 // type_checker.cpp
7729 // ::::::::::::::
7730 /*
7731 Copyright (c) 2013-14 Microsoft Corporation. All rights reserved.
7732 Released under Apache 2.0 license as described in the file LICENSE.
7733
7734 Author: Leonardo de Moura
7735 */
7736 #include <lean/flet.h>
7737 #include <lean/interrupt.h>
7738 #include <lean/sstream.h>
7739
7740 #include <utility>
7741 #include <vector>
7742
7743 #include "kernel/abstract.h"
7744 #include "kernel/expr_maps.h"
7745 #include "kernel/for_each_fn.h"
7746 #include "kernel/inductive.h"
7747 #include "kernel/instantiate.h"
7748 #include "kernel/kernel_exception.h"
7749 #include "kernel/quot.h"
7750 #include "kernel/replace_fn.h"
7751 #include "kernel/type_checker.h"
7752 #include "util/lbool.h"
7753
7754 namespace lean {
7755 static name *g_kernel_fresh = nullptr;
7756 static expr *g_dont_care = nullptr;
7757 static expr *g_nat_zero = nullptr;
7758 static expr *g_nat_succ = nullptr;
7759 static expr *g_nat_add = nullptr;
7760 static expr *g_nat_sub = nullptr;
7761 static expr *g_nat_mul = nullptr;
7762 static expr *g_nat_mod = nullptr;
7763 static expr *g_nat_div = nullptr;
7764 static expr *g_nat_beq = nullptr;
7765 static expr *g_nat_ble = nullptr;
7766
7767 type_checker::state::state(environment const &env)
7768     : m_env(env), mngen(*g_kernel_fresh) {}
7769

```

```

7770 /** \brief Make sure \c e "is" a sort, and return the corresponding sort.
7771     If \c e is not a sort, then the whnf procedure is invoked.
7772
7773     \remark \c s is used to extract position (line number information) when an
7774     error message is produced */
7775 expr type_checker::ensure_sort_core(expr e, expr const &s) {
7776     if (is_sort(e)) return e;
7777     auto new_e = whnf(e);
7778     if (is_sort(new_e)) {
7779         return new_e;
7780     } else {
7781         throw type_expected_exception(env(), m_lctx, s);
7782     }
7783 }
7784
7785 /** \brief Similar to \c ensure_sort, but makes sure \c e "is" a Pi. */
7786 expr type_checker::ensure_pi_core(expr e, expr const &s) {
7787     if (is_pi(e)) return e;
7788     auto new_e = whnf(e);
7789     if (is_pi(new_e)) {
7790         return new_e;
7791     } else {
7792         throw function_expected_exception(env(), m_lctx, s);
7793     }
7794 }
7795
7796 void type_checker::check_level(level const &l) {
7797     if (m_lparams) {
7798         if (auto n2 = get_undef_param(l, *m_lparams))
7799             throw kernel_exception(
7800                 env(), sstream() << "invalid reference to undefined universe "
7801                     "level parameter '"
7802                     << *n2 << "'");
7803     }
7804 }
7805
7806 expr type_checker::infer_fvar(expr const &e) {
7807     if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
7808         return decl->get_type();
7809     } else {
7810         throw kernel_exception(env(), "unknown free variable");
7811     }
7812 }
7813
7814 expr type_checker::infer_constant(expr const &e, bool infer_only) {
7815     constant_info info = env().get(const_name(e));
7816     auto const &ps = info.get_lparams();
7817     auto const &ls = const_levels(e);
7818     if (length(ps) != length(ls))
7819         throw kernel_exception(
7820             env(), sstream()
7821                 << "incorrect number of universe levels parameters for '"
7822                 << const_name(e) << "', #" << length(ps)
7823                 << " expected, #" << length(ls) << " provided");
7824     if (!infer_only) {
7825         if (m_safe_only && info.is_unsafe()) {
7826             throw kernel_exception(
7827                 env(),
7828                 sstream() << "invalid declaration, it uses unsafe declaration '"
7829                     << const_name(e) << "'");
7830         }
7831         for (level const &l : ls) check_level(l);
7832     }
7833     return instantiate_type_lparams(info, ls);
7834 }
7835
7836 expr type_checker::infer_lambda(expr const &e, bool infer_only) {
7837     flet<local_ctx> save_lctx(m_lctx, m_lctx);
7838     buffer<expr> fvars;
7839     expr e = _e;

```

```

7840     while (is_lambda(e)) {
7841         expr d = instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
7842         expr fvar = m_lctx.mk_local_decl(m_st->m_nngen, binding_name(e), d,
7843                                         binding_info(e));
7844         fvars.push_back(fvar);
7845         if (!infer_only) {
7846             ensure_sort_core(infer_type_core(d, infer_only), d);
7847         }
7848         e = binding_body(e);
7849     }
7850     expr r = infer_type_core(instantiate_rev(e, fvars.size(), fvars.data()),
7851                             infer_only);
7852     r = cheap_beta_reduce(r);
7853     return m_lctx.mk_pi(fvars, r);
7854 }
7855
7856 expr type_checker::infer_pi(expr const &e, bool infer_only) {
7857     flet<local_ctx> save_lctx(m_lctx, m_lctx);
7858     buffer<expr> fvars;
7859     buffer<level> us;
7860     expr e = _e;
7861     while (is_pi(e)) {
7862         expr d = instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
7863         expr t1 = ensure_sort_core(infer_type_core(d, infer_only), d);
7864         us.push_back(sort_level(t1));
7865         expr fvar = m_lctx.mk_local_decl(m_st->m_nngen, binding_name(e), d,
7866                                         binding_info(e));
7867         fvars.push_back(fvar);
7868         e = binding_body(e);
7869     }
7870     e = instantiate_rev(e, fvars.size(), fvars.data());
7871     expr s = ensure_sort_core(infer_type_core(e, infer_only), e);
7872     level r = sort_level(s);
7873     unsigned i = fvars.size();
7874     while (i > 0) {
7875         --i;
7876         r = mk_imax(us[i], r);
7877     }
7878     return mk_sort(r);
7879 }
7880
7881 expr type_checker::infer_app(expr const &e, bool infer_only) {
7882     if (!infer_only) {
7883         expr f_type = ensure_pi_core(infer_type_core(app_fn(e), infer_only), e);
7884         expr a_type = infer_type_core(app_arg(e), infer_only);
7885         expr d_type = binding_domain(f_type);
7886         if (!is_def_eq(a_type, d_type)) {
7887             throw app_type_mismatch_exception(env(), m_lctx, e, f_type, a_type);
7888         }
7889         return instantiate(binding_body(f_type), app_arg(e));
7890     } else {
7891         buffer<expr> args;
7892         expr const &f = get_app_args(e, args);
7893         expr f_type = infer_type_core(f, true);
7894         unsigned j = 0;
7895         unsigned nargs = args.size();
7896         for (unsigned i = 0; i < nargs; i++) {
7897             if (is_pi(f_type)) {
7898                 f_type = binding_body(f_type);
7899             } else {
7900                 f_type = instantiate_rev(f_type, i - j, args.data() + j);
7901                 f_type = ensure_pi_core(f_type, e);
7902                 f_type = binding_body(f_type);
7903                 j = i;
7904             }
7905         }
7906         return instantiate_rev(f_type, nargs - j, args.data() + j);
7907     }
7908 }
7909

```

```

7910 static void mark_used(unsigned n, expr const *fvars, expr const &b,
7911                        bool *used) {
7912     if (!has_fvar(b)) return;
7913     for_each(b, [&](expr const &x, unsigned) {
7914         if (!has_fvar(x)) return false;
7915         if (is_fvar(x)) {
7916             for (unsigned i = 0; i < n; i++) {
7917                 if (fvar_name(fvars[i]) == fvar_name(x)) {
7918                     used[i] = true;
7919                     return false;
7920                 }
7921             }
7922         }
7923     });
7924     return true;
7925 }
7926
7927 expr type_checker::infer_let(expr const &e, bool infer_only) {
7928     flet<local_ctx> save_lctx(m_lctx, m_lctx);
7929     buffer<expr> fvars;
7930     buffer<expr> vals;
7931     expr e = _e;
7932     while (is_let(e)) {
7933         expr type = instantiate_rev(let_type(e), fvars.size(), fvars.data());
7934         expr val = instantiate_rev(let_value(e), fvars.size(), fvars.data());
7935         expr fvar = m_lctx.mk_local_decl(m_st->m_nngen, let_name(e), type, val);
7936         fvars.push_back(fvar);
7937         vals.push_back(val);
7938         if (!infer_only) {
7939             ensure_sort_core(infer_type_core(type, infer_only), type);
7940             expr val_type = infer_type_core(val, infer_only);
7941             if (!is_def_eq(val_type, type)) {
7942                 throw def_type_mismatch_exception(env(), m_lctx, let_name(e),
7943                                                    val_type, type);
7944             }
7945         }
7946         e = let_body(e);
7947     }
7948     expr r = infer_type_core(instantiate_rev(e, fvars.size(), fvars.data()),
7949                             infer_only);
7950     r = cheap_beta_reduce(r); // use `cheap_beta_reduce` (to try) to reduce
7951                             // number of dependencies
7952     buffer<bool, 128> used;
7953     used.resize(fvars.size(), false);
7954     mark_used(fvars.size(), fvars.data(), r, used.data());
7955     unsigned i = fvars.size();
7956     while (i > 0) {
7957         --i;
7958         if (used[i]) mark_used(i, fvars.data(), vals[i], used.data());
7959     }
7960     buffer<expr> used_fvars;
7961     for (unsigned i = 0; i < fvars.size(); i++) {
7962         if (used[i]) used_fvars.push_back(fvars[i]);
7963     }
7964     return m_lctx.mk_pi(used_fvars, r);
7965 }
7966
7967 expr type_checker::infer_proj(expr const &e, bool infer_only) {
7968     expr type = whnf(infer_type_core(proj_expr(e), infer_only));
7969     if (!proj_idx(e).is_small()) throw invalid_proj_exception(env(), m_lctx, e);
7970     unsigned idx = proj_idx(e).get_small_value();
7971     buffer<expr> args;
7972     expr const &I = get_app_args(type, args);
7973     if (!is_constant(I)) throw invalid_proj_exception(env(), m_lctx, e);
7974     name const &I_name = const_name(I);
7975     if (I_name != proj_sname(e)) throw invalid_proj_exception(env(), m_lctx, e);
7976     constant_info I_info = env().get(I_name);
7977     if (!I_info.is_inductive()) throw invalid_proj_exception(env(), m_lctx, e);
7978     inductive_val I_val = I_info.to_inductive_val();
7979     if (length(I_val.get_cnstrs()) != 1 ||

```



```

7980     args.size() != I_val.get_nparams() + I_val.get_nindices())
7981     throw invalid_proj_exception(env(), m_lctx, e);
7982
7983     constant_info c_info = env().get(head(I_val.get_cnstrs()));
7984     expr r = instantiate_type_lparams(c_info, const_levels(I));
7985     for (unsigned i = 0; i < I_val.get_nparams(); i++) {
7986         lean_assert(i < args.size());
7987         r = whnf(r);
7988         if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
7989         r = instantiate(binding_body(r), args[i]);
7990     }
7991     for (unsigned i = 0; i < idx; i++) {
7992         r = whnf(r);
7993         if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
7994         if (has_loose_bvars(binding_body(r)))
7995             r = instantiate(binding_body(r), mk_proj(I_name, i, proj_expr(e)));
7996         else
7997             r = binding_body(r);
7998     }
7999     r = whnf(r);
8000     if (!is_pi(r)) throw invalid_proj_exception(env(), m_lctx, e);
8001     return binding_domain(r);
8002 }
8003
8004 /** \brief Return type of expression \c e, if \c infer_only is false, then it
8005     also check whether \c e is type correct or not. \pre closed(e) */
8006 expr type_checker::infer_type_core(expr const &e, bool infer_only) {
8007     if (is_bvar(e))
8008         throw kernel_exception(
8009             env(),
8010             "type checker does not support loose bound variables, replace them "
8011             "with free variables before invoking it");
8012
8013     lean_assert(!has_loose_bvars(e));
8014     check_system("type checker");
8015
8016     auto it = m_st->m_infer_type[infer_only].find(e);
8017     if (it != m_st->m_infer_type[infer_only].end()) return it->second;
8018
8019     expr r;
8020     switch (e.kind()) {
8021     case expr_kind::Lit:
8022         r = lit_type(lit_value(e));
8023         break;
8024     case expr_kind::MData:
8025         r = infer_type_core(mdata_expr(e), infer_only);
8026         break;
8027     case expr_kind::Proj:
8028         r = infer_proj(e, infer_only);
8029         break;
8030     case expr_kind::FVar:
8031         r = infer_fvar(e);
8032         break;
8033     case expr_kind::MVar:
8034         throw kernel_exception(
8035             env(), "kernel type checker does not support meta variables");
8036     case expr_kind::BVar:
8037         lean_unreachable(); // LCOV_EXCL_LINE
8038     case expr_kind::Sort:
8039         if (!infer_only) check_level(sort_level(e));
8040         r = mk_sort(mk_succ(sort_level(e)));
8041         break;
8042     case expr_kind::Const:
8043         r = infer_constant(e, infer_only);
8044         break;
8045     case expr_kind::Lambda:
8046         r = infer_lambda(e, infer_only);
8047         break;
8048     case expr_kind::Pi:
8049         r = infer_pi(e, infer_only);

```

```

8050         break;
8051     case expr_kind::App:
8052         r = infer_app(e, infer_only);
8053         break;
8054     case expr_kind::Let:
8055         r = infer_let(e, infer_only);
8056         break;
8057 }
8058
8059 m_st->m_infer_type[infer_only].insert(mk_pair(e, r));
8060 return r;
8061 }
8062
8063 expr type_checker::infer_type(expr const &e) {
8064     return infer_type_core(e, true);
8065 }
8066
8067 expr type_checker::check(expr const &e, names const &lps) {
8068     flet<names const *> updt(m_lparams, &lps);
8069     return infer_type_core(e, false);
8070 }
8071
8072 expr type_checker::check_ignore_undefined_universes(expr const &e) {
8073     flet<names const *> updt(m_lparams, nullptr);
8074     return infer_type_core(e, false);
8075 }
8076
8077 expr type_checker::ensure_sort(expr const &e, expr const &s) {
8078     return ensure_sort_core(e, s);
8079 }
8080
8081 expr type_checker::ensure_pi(expr const &e, expr const &s) {
8082     return ensure_pi_core(e, s);
8083 }
8084
8085 /** \brief Return true iff \c e is a proposition */
8086 bool type_checker::is_prop(expr const &e) {
8087     return whnf(infer_type(e)) == mk_Prop();
8088 }
8089
8090 /** \brief Apply normalizer extensions to \c e.
8091     If `cheap == true`, then we don't perform delta-reduction when reducing
8092     major premise. */
8093 optional<expr> type_checker::reduce_recursor(expr const &e, bool cheap) {
8094     if (env().is_quot_initialized()) {
8095         if (optional<expr> r =
8096             quot_reduce_rec(e, [&](expr const &e) { return whnf(e); })) {
8097             return r;
8098         }
8099     }
8100     if (optional<expr> r = inductive_reduce_rec(
8101         env(), e,
8102         [&](expr const &e) {
8103             return cheap ? whnf_core(e, cheap) : whnf(e);
8104         },
8105         [&](expr const &e) { return infer(e); },
8106         [&](expr const &e1, expr const &e2) {
8107             return is_def_eq(e1, e2);
8108         }
8109     )) {
8110         return r;
8111     }
8112     return none_expr();
8113 }
8114
8115 expr type_checker::whnf_fvar(expr const &e, bool cheap) {
8116     if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
8117         if (optional<expr> const &v = decl->get_value()) {
8118             /* zeta-reduction */
8119             return whnf_core(*v, cheap);
8120         }
8121     }

```

```

8120     }
8121     return e;
8122 }
8123
8124 /* If `cheap == true`, then we don't perform delta-reduction when reducing major
8125  * premise. */
8126 optional<expr> type_checker::reduce_proj(expr const &e, bool cheap) {
8127     if (!proj_idx(e).is_small()) return none_expr();
8128     unsigned idx = proj_idx(e).get_small_value();
8129     expr c;
8130     if (cheap)
8131         c = whnf_core(proj_expr(e), cheap);
8132     else
8133         c = whnf(proj_expr(e));
8134     buffer<expr> args;
8135     expr const &mk = get_app_args(c, args);
8136     if (!is_constant(mk)) return none_expr();
8137     constant_info mk_info = env().get(const_name(mk));
8138     if (!mk_info.is_constructor()) return none_expr();
8139     unsigned nparams = mk_info.to_constructor_val().get_nparams();
8140     if (nparams + idx < args.size())
8141         return some_expr(args[nparams + idx]);
8142     else
8143         return none_expr();
8144 }
8145
8146 static bool is_let_fvar(local_ctx const &lctx, expr const &e) {
8147     lean_assert(is_fvar(e));
8148     if (optional<local_decl> decl = lctx.find_local_decl(e)) {
8149         return static_cast<bool>(decl->get_value());
8150     } else {
8151         return false;
8152     }
8153 }
8154
8155 /** \brief Weak head normal form core procedure. It does not perform delta
8156     reduction nor normalization extensions. If `cheap == true`, then we don't
8157     perform delta-reduction when reducing major premise of recursors and
8158     projections. We also do not cache results. */
8159 expr type_checker::whnf_core(expr const &e, bool cheap) {
8160     check_system("whnf");
8161
8162     // handle easy cases
8163     switch (e.kind()) {
8164     case expr_kind::BVar:
8165     case expr_kind::Sort:
8166     case expr_kind::MVar:
8167     case expr_kind::Pi:
8168     case expr_kind::Const:
8169     case expr_kind::Lambda:
8170     case expr_kind::Lit:
8171         return e;
8172     case expr_kind::MData:
8173         return whnf_core(mdata_expr(e), cheap);
8174     case expr_kind::FVar:
8175         if (is_let_fvar(m_lctx, e))
8176             break;
8177         else
8178             return e;
8179     case expr_kind::App:
8180     case expr_kind::Let:
8181     case expr_kind::Proj:
8182         break;
8183     }
8184
8185     // check cache
8186     if (!cheap) {
8187         auto it = m_st->m_whnf_core.find(e);
8188         if (it != m_st->m_whnf_core.end()) return it->second;
8189     }

```

```

8190
8191 // do the actual work
8192 expr r;
8193 switch (e.kind()) {
8194     case expr_kind::BVar:
8195     case expr_kind::Sort:
8196     case expr_kind::MVar:
8197     case expr_kind::Pi:
8198     case expr_kind::Const:
8199     case expr_kind::Lambda:
8200     case expr_kind::Lit:
8201     case expr_kind::MData:
8202         lean_unreachable(); // LCOV_EXCL_LINE
8203     case expr_kind::FVar:
8204         return whnf_fvar(e, cheap);
8205     case expr_kind::Proj: {
8206         if (auto m = reduce_proj(e, cheap))
8207             r = whnf_core(*m, cheap);
8208         else
8209             r = e;
8210         break;
8211     }
8212     case expr_kind::App: {
8213         buffer<expr> args;
8214         expr f0 = get_app_rev_args(e, args);
8215         expr f = whnf_core(f0, cheap);
8216         if (is_lambda(f)) {
8217             unsigned m = 1;
8218             unsigned num_args = args.size();
8219             while (is_lambda(binding_body(f)) && m < num_args) {
8220                 f = binding_body(f);
8221                 m++;
8222             }
8223             lean_assert(m <= num_args);
8224             r = whnf_core(
8225                 mk_rev_app(instantiate(binding_body(f), m,
8226                                     args.data() + (num_args - m)),
8227                             num_args - m, args.data()),
8228                             cheap);
8229         } else if (f == f0) {
8230             if (auto r = reduce_recursor(e, cheap)) {
8231                 /* iota-reduction and quotient reduction rules */
8232                 return whnf_core(*r, cheap);
8233             } else {
8234                 return e;
8235             }
8236         } else {
8237             r = whnf_core(mk_rev_app(f, args.size(), args.data()), cheap);
8238         }
8239         break;
8240     }
8241     case expr_kind::Let:
8242         r = whnf_core(instantiate(let_body(e), let_value(e)), cheap);
8243         break;
8244 }
8245
8246 if (!cheap) {
8247     m_st->m_whnf_core.insert(mk_pair(e, r));
8248 }
8249 return r;
8250 }
8251
8252 /** \brief Return some definition \c d iff \c e is a target for delta-reduction,
8253     and the given definition is the one to be expanded. */
8254 optional<constant_info> type_checker::is_delta(expr const &e) const {
8255     expr const &f = get_app_fn(e);
8256     if (is_constant(f)) {
8257         if (optional<constant_info> info = env().find(const_name(f)))
8258             if (info->has_value()) return info;
8259     }

```

```

8260     return none_constant_info();
8261 }
8262
8263 optional<expr> type_checker::unfold_definition_core(expr const &e) {
8264     if (is_constant(e)) {
8265         if (auto d = is_delta(e)) {
8266             if (length(const_levels(e)) == d->get_num_lparams())
8267                 return some_expr(
8268                     instantiate_value_lparams(*d, const_levels(e)));
8269         }
8270     }
8271     return none_expr();
8272 }
8273
8274 /* Unfold head(e) if it is a constant */
8275 optional<expr> type_checker::unfold_definition(expr const &e) {
8276     if (is_app(e)) {
8277         expr f0 = get_app_fn(e);
8278         if (auto f = unfold_definition_core(f0)) {
8279             buffer<expr> args;
8280             get_app_rev_args(e, args);
8281             return some_expr(mk_rev_app(*f, args));
8282         } else {
8283             return none_expr();
8284         }
8285     } else {
8286         return unfold_definition_core(e);
8287     }
8288 }
8289
8290 static expr *g_lean_reduce_bool = nullptr;
8291 static expr *g_lean_reduce_nat = nullptr;
8292
8293 namespace ir {
8294 object *run_boxed(environment const &env, options const &opts, name const &fn,
8295                 unsigned n, object **args);
8296 }
8297
8298 expr mk_bool_true();
8299 expr mk_bool_false();
8300
8301 optional<expr> reduce_native(environment const &env, expr const &e) {
8302     if (!is_app(e)) return none_expr();
8303     expr const &arg = app_arg(e);
8304     if (!is_constant(arg)) return none_expr();
8305     if (app_fn(e) == *g_lean_reduce_bool) {
8306         object *r = ir::run_boxed(env, options(), const_name(arg), 0, nullptr);
8307         if (!lean_is_scalar(r)) {
8308             lean_dec_ref(r);
8309             throw kernel_exception(env,
8310                 "type checker failure, unexpected result "
8311                 "value for 'Lean.reduceBool'");
8312         }
8313         return lean_unbox(r) == 0 ? some_expr(mk_bool_false())
8314             : some_expr(mk_bool_true());
8315     }
8316     if (app_fn(e) == *g_lean_reduce_nat) {
8317         object *r = ir::run_boxed(env, options(), const_name(arg), 0, nullptr);
8318         if (lean_is_scalar(r) || lean_is_mpz(r)) {
8319             return some_expr(mk_lit(literal(nat(r))));
8320         } else {
8321             throw kernel_exception(env,
8322                 "type checker failure, unexpected result "
8323                 "value for 'Lean.reduceNat'");
8324         }
8325     }
8326     return none_expr();
8327 }
8328
8329 static inline bool is_nat_lit_ext(expr const &e) {

```

```

8330     return e == *g_nat_zero || is_nat_lit(e);
8331 }
8332 static inline nat get_nat_val(expr const &e) {
8333     lean_assert(is_nat_lit_ext(e));
8334     if (e == *g_nat_zero) return nat((unsigned)0);
8335     return lit_value(e).get_nat();
8336 }
8337
8338 template <typename F>
8339 optional<expr> type_checker::reduce_bin_nat_op(F const &f, expr const &e) {
8340     expr arg1 = whnf(app_arg(app_fn(e)));
8341     if (!is_nat_lit_ext(arg1)) return none_expr();
8342     expr arg2 = whnf(app_arg(e));
8343     if (!is_nat_lit_ext(arg2)) return none_expr();
8344     nat v1 = get_nat_val(arg1);
8345     nat v2 = get_nat_val(arg2);
8346     return some_expr(mk_lit(literal(nat(f(v1.raw(), v2.raw())))));
8347 }
8348
8349 template <typename F>
8350 optional<expr> type_checker::reduce_bin_nat_pred(F const &f, expr const &e) {
8351     expr arg1 = whnf(app_arg(app_fn(e)));
8352     if (!is_nat_lit_ext(arg1)) return none_expr();
8353     expr arg2 = whnf(app_arg(e));
8354     if (!is_nat_lit_ext(arg2)) return none_expr();
8355     nat v1 = get_nat_val(arg1);
8356     nat v2 = get_nat_val(arg2);
8357     return f(v1.raw(), v2.raw()) ? some_expr(mk_bool_true())
8358                                   : some_expr(mk_bool_false());
8359 }
8360
8361 optional<expr> type_checker::reduce_nat(expr const &e) {
8362     if (has_fvar(e)) return none_expr();
8363     unsigned nargs = get_app_num_args(e);
8364     if (nargs == 1) {
8365         expr const &f = app_fn(e);
8366         if (f == *g_nat_succ) {
8367             expr arg = whnf(app_arg(e));
8368             if (!is_nat_lit_ext(arg)) return none_expr();
8369             nat v = get_nat_val(arg);
8370             return some_expr(mk_lit(literal(nat(v + nat(1)))));
8371         }
8372     } else if (nargs == 2) {
8373         expr const &f = app_fn(app_fn(e));
8374         if (!is_constant(f)) return none_expr();
8375         if (f == *g_nat_add) return reduce_bin_nat_op(nat_add, e);
8376         if (f == *g_nat_sub) return reduce_bin_nat_op(nat_sub, e);
8377         if (f == *g_nat_mul) return reduce_bin_nat_op(nat_mul, e);
8378         if (f == *g_nat_mod) return reduce_bin_nat_op(nat_mod, e);
8379         if (f == *g_nat_div) return reduce_bin_nat_op(nat_div, e);
8380         if (f == *g_nat_beq) return reduce_bin_nat_pred(nat_eq, e);
8381         if (f == *g_nat_ble) return reduce_bin_nat_pred(nat_le, e);
8382     }
8383     return none_expr();
8384 }
8385
8386 /** \brief Put expression \c t in weak head normal form */
8387 expr type_checker::whnf(expr const &e) {
8388     // Do not cache easy cases
8389     switch (e.kind()) {
8390     case expr_kind::BVar:
8391     case expr_kind::Sort:
8392     case expr_kind::MVar:
8393     case expr_kind::Pi:
8394     case expr_kind::Lit:
8395         return e;
8396     case expr_kind::MData:
8397         return whnf(mdata_expr(e));
8398     case expr_kind::FVar:
8399         if (is_let_fvar(m_lctx, e))

```

```

8400         break;
8401     else
8402         return e;
8403     case expr_kind::Lambda:
8404     case expr_kind::App:
8405     case expr_kind::Const:
8406     case expr_kind::Let:
8407     case expr_kind::Proj:
8408         break;
8409 }
8410
8411 // check cache
8412 auto it = m_st->m_wnhf.find(e);
8413 if (it != m_st->m_wnhf.end()) return it->second;
8414
8415 expr t = e;
8416 while (true) {
8417     expr t1 = whnf_core(t);
8418     if (auto v = reduce_native(env(), t1)) {
8419         m_st->m_wnhf.insert(mk_pair(e, *v));
8420         return *v;
8421     } else if (auto v = reduce_nat(t1)) {
8422         m_st->m_wnhf.insert(mk_pair(e, *v));
8423         return *v;
8424     } else if (auto next_t = unfold_definition(t1)) {
8425         t = *next_t;
8426     } else {
8427         auto r = t1;
8428         m_st->m_wnhf.insert(mk_pair(e, r));
8429         return r;
8430     }
8431 }
8432 }
8433
8434 /** \brief Given lambda/Pi expressions \c t and \c s, return true iff \c t is
8435     def eq to \c s.
8436
8437     t and s are definitionally equal
8438     iff
8439     domain(t) is definitionally equal to domain(s)
8440     and
8441     body(t) is definitionally equal to body(s) */
8442 bool type_checker::is_def_eq_binding(expr t, expr s) {
8443     lean_assert(t.kind() == s.kind());
8444     lean_assert(is_binding(t));
8445     flet<local_ctx> save_lctx(m_lctx, m_lctx);
8446     expr_kind k = t.kind();
8447     buffer<expr> subst;
8448     do {
8449         optional<expr> var_s_type;
8450         if (binding_domain(t) != binding_domain(s)) {
8451             var_s_type =
8452                 instantiate_rev(binding_domain(s), subst.size(), subst.data());
8453             expr var_t_type =
8454                 instantiate_rev(binding_domain(t), subst.size(), subst.data());
8455             if (!is_def_eq(var_t_type, *var_s_type)) return false;
8456         }
8457         if (has_loose_bvars(binding_body(t)) ||
8458             has_loose_bvars(binding_body(s))) {
8459             // free variable is used inside t or s
8460             if (!var_s_type)
8461                 var_s_type = instantiate_rev(binding_domain(s), subst.size(),
8462                                                 subst.data());
8463             subst.push_back(m_lctx.mk_local_decl(m_st->m_nngen, binding_name(s),
8464                                                 *var_s_type, binding_info(s)));
8465         } else {
8466             subst.push_back(*g_dont_care); // don't care
8467         }
8468         t = binding_body(t);
8469         s = binding_body(s);

```



```

8470     } while (t.kind() == k && s.kind() == k);
8471     return is_def_eq(instantiate_rev(t, subst.size(), subst.data()),
8472                     instantiate_rev(s, subst.size(), subst.data()));
8473 }
8474
8475 bool type_checker::is_def_eq(level const &l1, level const &l2) {
8476     if (is_equivalent(l1, l2)) {
8477         return true;
8478     } else {
8479         return false;
8480     }
8481 }
8482
8483 bool type_checker::is_def_eq(levels const &ls1, levels const &ls2) {
8484     if (is_nil(ls1) && is_nil(ls2)) {
8485         return true;
8486     } else if (!is_nil(ls1) && !is_nil(ls2)) {
8487         return is_def_eq(head(ls1), head(ls2)) &&
8488                is_def_eq(tail(ls1), tail(ls2));
8489     } else {
8490         return false;
8491     }
8492 }
8493
8494 /** \brief This is an auxiliary method for is_def_eq. It handles the "easy
8495     * cases". */
8496 lbool type_checker::quick_is_def_eq(expr const &t, expr const &s,
8497                                     bool use_hash) {
8498     if (m_st->m_eqv_manager.is_equiv(t, s, use_hash)) return l_true;
8499     if (t.kind() == s.kind()) {
8500         switch (t.kind()) {
8501             case expr_kind::Lambda:
8502             case expr_kind::Pi:
8503                 return to_lbool(is_def_eq_binding(t, s));
8504             case expr_kind::Sort:
8505                 return to_lbool(is_def_eq(sort_level(t), sort_level(s)));
8506             case expr_kind::MData:
8507                 return to_lbool(is_def_eq(mdata_expr(t), mdata_expr(s)));
8508             case expr_kind::MVar:
8509                 lean_unreachable(); // LCOV_EXCL_LINE
8510             case expr_kind::BVar:
8511             case expr_kind::FVar:
8512             case expr_kind::App:
8513             case expr_kind::Const:
8514             case expr_kind::Let:
8515             case expr_kind::Proj:
8516                 // We do not handle these cases in this method.
8517                 break;
8518             case expr_kind::Lit:
8519                 return to_lbool(lit_value(t) == lit_value(s));
8520         }
8521     }
8522     return l_undef; // This is not an "easy case"
8523 }
8524
8525 /** \brief Return true if arguments of \c t are definitionally equal to
8526     arguments of \c s. This method is used to implement an optimization in the
8527     method \c is_def_eq. */
8528 bool type_checker::is_def_eq_args(expr t, expr s) {
8529     while (is_app(t) && is_app(s)) {
8530         if (!is_def_eq(app_arg(t), app_arg(s))) return false;
8531         t = app_fn(t);
8532         s = app_fn(s);
8533     }
8534     return !is_app(t) && !is_app(s);
8535 }
8536
8537 /** \brief Try to solve (fun (x : A), B) =?= s by trying eta-expansion on s */
8538 bool type_checker::try_eta_expansion_core(expr const &t, expr const &s) {
8539     if (is_lambda(t) && !is_lambda(s)) {

```

```

8540     expr s_type = whnf(infer_type(s));
8541     if (!is_pi(s_type)) return false;
8542     expr new_s = mk_lambda(binding_name(s_type), binding_domain(s_type),
8543                           mk_app(s, mk_bvar(0)), binding_info(s_type));
8544     if (!is_def_eq(t, new_s)) return false;
8545     return true;
8546 } else {
8547     return false;
8548 }
8549 }
8550
8551 /** \brief Return true if \c t and \c s are definitionally equal because they
8552     are applications of the form <tt>(f a_1 ... a_n)</tt> <tt>(g b_1 ...
8553     b_n)</tt>, and \c f and \c g are definitionally equal, and \c a_i and \c b_i
8554     are also definitionally equal for every 1 <= i <= n.
8555     Return false otherwise. */
8556 bool type_checker::is_def_eq_app(expr const &t, expr const &s) {
8557     if (is_app(t) && is_app(s)) {
8558         buffer<expr> t_args;
8559         buffer<expr> s_args;
8560         expr t_fn = get_app_args(t, t_args);
8561         expr s_fn = get_app_args(s, s_args);
8562         if (is_def_eq(t_fn, s_fn) && t_args.size() == s_args.size()) {
8563             unsigned i = 0;
8564             for (; i < t_args.size(); i++) {
8565                 if (!is_def_eq(t_args[i], s_args[i])) break;
8566             }
8567             if (i == t_args.size()) return true;
8568         }
8569     }
8570     return false;
8571 }
8572
8573 /** \brief Return true if \c t and \c s are definitionally equal due to proof
8574     irrelevant. Return false otherwise. */
8575 bool type_checker::is_def_eq_proof_irrel(expr const &t, expr const &s) {
8576     // Proof irrelevance support for Prop (aka Type.{0})
8577     expr t_type = infer_type(t);
8578     if (!is_prop(t_type)) return false;
8579     expr s_type = infer_type(s);
8580     return is_def_eq(t_type, s_type);
8581 }
8582
8583 bool type_checker::failed_before(expr const &t, expr const &s) const {
8584     if (hash(t) < hash(s)) {
8585         return m_st->m_failure.find(mk_pair(t, s)) != m_st->m_failure.end();
8586     } else if (hash(t) > hash(s)) {
8587         return m_st->m_failure.find(mk_pair(s, t)) != m_st->m_failure.end();
8588     } else {
8589         return m_st->m_failure.find(mk_pair(t, s)) != m_st->m_failure.end() ||
8590             m_st->m_failure.find(mk_pair(s, t)) != m_st->m_failure.end();
8591     }
8592 }
8593
8594 void type_checker::cache_failure(expr const &t, expr const &s) {
8595     if (hash(t) <= hash(s))
8596         m_st->m_failure.insert(mk_pair(t, s));
8597     else
8598         m_st->m_failure.insert(mk_pair(s, t));
8599 }
8600
8601 /** \brief Perform one lazy delta-reduction step.
8602     Return
8603     - l_true if t_n and s_n are definitionally equal.
8604     - l_false if they are not definitionally equal.
8605     - l_undef if the step did not manage to establish whether they are
8606     definitionally equal or not.
8607
8608     \remark t_n, s_n and cs are updated. */
8609 auto type_checker::lazy_delta_reduction_step(expr &t_n, expr &s_n)

```

```

8610     -> reduction_status {
8611     auto d_t = is_delta(t_n);
8612     auto d_s = is_delta(s_n);
8613     if (!d_t && !d_s) {
8614         return reduction_status::DefUnknown;
8615     } else if (d_t && !d_s) {
8616         t_n = whnf_core(*unfold_definition(t_n));
8617     } else if (!d_t && d_s) {
8618         s_n = whnf_core(*unfold_definition(s_n));
8619     } else {
8620         int c = compare(d_t->get_hints(), d_s->get_hints());
8621         if (c < 0) {
8622             t_n = whnf_core(*unfold_definition(t_n));
8623         } else if (c > 0) {
8624             s_n = whnf_core(*unfold_definition(s_n));
8625         } else {
8626             if (is_app(t_n) && is_app(s_n) && is_eqp(*d_t, *d_s)) {
8627                 // Optimization:
8628                 // We try to check if their arguments are definitionally equal.
8629                 // If they are, then t_n and s_n must be definitionally equal,
8630                 // and we can skip the delta-reduction step.
8631                 if (!failed_before(t_n, s_n)) {
8632                     if (is_def_eq(const_levels(get_app_fn(t_n)),
8633                                   const_levels(get_app_fn(s_n))) &&
8634                         is_def_eq_args(t_n, s_n)) {
8635                         return reduction_status::DefEqual;
8636                     } else {
8637                         cache_failure(t_n, s_n);
8638                     }
8639                 }
8640             }
8641             t_n = whnf_core(*unfold_definition(t_n));
8642             s_n = whnf_core(*unfold_definition(s_n));
8643         }
8644     }
8645     switch (quick_is_def_eq(t_n, s_n)) {
8646     case l_true:
8647         return reduction_status::DefEqual;
8648     case l_false:
8649         return reduction_status::DefDiff;
8650     case l_undef:
8651         return reduction_status::Continue;
8652     }
8653     lean_unreachable();
8654 }
8655
8656 inline bool is_nat_zero(expr const &t) {
8657     return t == *g_nat_zero || (is_nat_lit(t) && lit_value(t).is_zero());
8658 }
8659
8660 inline optional<expr> is_nat_succ(expr const &t) {
8661     if (is_nat_lit(t)) {
8662         nat val = lit_value(t).get_nat();
8663         if (!val.is_zero()) {
8664             return some_expr(mk_lit(literal(val - nat(1))));
8665         }
8666     }
8667
8668     if (get_app_fn(t) == *g_nat_succ && get_app_num_args(t) == 1) {
8669         return some_expr(app_arg(t));
8670     }
8671     return none_expr();
8672 }
8673
8674 lbool type_checker::is_def_eq_offset(expr const &t, expr const &s) {
8675     if (is_nat_zero(t) && is_nat_zero(s)) return l_true;
8676     optional<expr> pred_t = is_nat_succ(t);
8677     optional<expr> pred_s = is_nat_succ(s);
8678     if (pred_t && pred_s) {
8679         return to_lbool(is_def_eq_core(*pred_t, *pred_s));
8680     }

```

```

8680     }
8681     return l_undef;
8682 }
8683
8684 lbool type_checker::lazy_delta_reduction(expr &t_n, expr &s_n) {
8685     while (true) {
8686         lbool r = is_def_eq_offset(t_n, s_n);
8687         if (r != l_undef) return r;
8688
8689         if (!has_fvar(t_n) && !has_fvar(s_n)) {
8690             if (auto t_v = reduce_nat(t_n)) {
8691                 return to_lbool(is_def_eq_core(*t_v, s_n));
8692             } else if (auto s_v = reduce_nat(s_n)) {
8693                 return to_lbool(is_def_eq_core(t_n, *s_v));
8694             }
8695         }
8696
8697         if (auto t_v = reduce_native(env(), t_n)) {
8698             return to_lbool(is_def_eq_core(*t_v, s_n));
8699         } else if (auto s_v = reduce_native(env(), s_n)) {
8700             return to_lbool(is_def_eq_core(t_n, *s_v));
8701         }
8702
8703         switch (lazy_delta_reduction_step(t_n, s_n)) {
8704             case reduction_status::Continue:
8705                 break;
8706             case reduction_status::DefUnknown:
8707                 return l_undef;
8708             case reduction_status::DefEqual:
8709                 return l_true;
8710             case reduction_status::DefDiff:
8711                 return l_false;
8712         }
8713     }
8714 }
8715
8716 static expr *g_string_mk = nullptr;
8717
8718 lbool type_checker::try_string_lit_expansion_core(expr const &t,
8719                                                    expr const &s) {
8720     if (is_string_lit(t) && is_app(s) && app_fn(s) == *g_string_mk) {
8721         return to_lbool(is_def_eq_core(string_lit_to_constructor(t), s));
8722     }
8723     return l_undef;
8724 }
8725
8726 lbool type_checker::try_string_lit_expansion(expr const &t, expr const &s) {
8727     lbool r = try_string_lit_expansion_core(t, s);
8728     if (r != l_undef) return r;
8729     return try_string_lit_expansion_core(s, t);
8730 }
8731
8732 bool type_checker::is_def_eq_core(expr const &t, expr const &s) {
8733     check_system("is_definitionally_equal");
8734     bool use_hash = true;
8735     lbool r = quick_is_def_eq(t, s, use_hash);
8736     if (r != l_undef) return r == l_true;
8737
8738     // apply whnf (without using delta-reduction or normalizer extensions)
8739     expr t_n = whnf_core(t);
8740     expr s_n = whnf_core(s);
8741
8742     if (!is_eqp(t_n, t) || !is_eqp(s_n, s)) {
8743         r = quick_is_def_eq(t_n, s_n);
8744         if (r != l_undef) return r == l_true;
8745     }
8746
8747     if (is_def_eq_proof_irrel(t_n, s_n)) return true;
8748
8749     r = lazy_delta_reduction(t_n, s_n);

```

```

8750     if (r != l_undef) return r == l_true;
8751
8752     if (is_constant(t_n) && is_constant(s_n) &&
8753         const_name(t_n) == const_name(s_n) &&
8754         is_def_eq(const_levels(t_n), const_levels(s_n)))
8755         return true;
8756
8757     if (is_fvar(t_n) && is_fvar(s_n) && fvar_name(t_n) == fvar_name(s_n))
8758         return true;
8759
8760     if (is_proj(t_n) && is_proj(s_n) && proj_idx(t_n) == proj_idx(s_n) &&
8761         is_def_eq(proj_expr(t_n), proj_expr(s_n)))
8762         return true;
8763
8764     // At this point, t_n and s_n are in weak head normal form (modulo
8765     // metavariables and proof irrelevance)
8766     if (is_def_eq_app(t_n, s_n)) return true;
8767
8768     if (try_eta_expansion(t_n, s_n)) return true;
8769
8770     r = try_string_lit_expansion(t_n, s_n);
8771     if (r != l_undef) return r == l_true;
8772
8773     return false;
8774 }
8775
8776 bool type_checker::is_def_eq(expr const &t, expr const &s) {
8777     bool r = is_def_eq_core(t, s);
8778     if (r) m_st->m_eqv_manager.add_equiv(t, s);
8779     return r;
8780 }
8781
8782 expr type_checker::eta_expand(expr const &e) {
8783     buffer<expr> fvars;
8784     flet<local_ctx> save_lctx(m_lctx, m_lctx);
8785     expr it = e;
8786     while (is_lambda(it)) {
8787         expr d =
8788             instantiate_rev(binding_domain(it), fvars.size(), fvars.data());
8789         fvars.push_back(m_lctx.mk_local_decl(m_st->m_ngen, binding_name(it), d,
8790             binding_info(it)));
8791         it = binding_body(it);
8792     }
8793     it = instantiate_rev(it, fvars.size(), fvars.data());
8794     expr it_type = whnf(infer(it));
8795     if (!is_pi(it_type)) return e;
8796     buffer<expr> args;
8797     while (is_pi(it_type)) {
8798         expr arg = m_lctx.mk_local_decl(m_st->m_ngen, binding_name(it_type),
8799             binding_domain(it_type),
8800             binding_info(it_type));
8801         args.push_back(arg);
8802         fvars.push_back(arg);
8803         it_type = whnf(instantiate(binding_body(it_type), arg));
8804     }
8805     expr r = mk_app(it, args);
8806     return m_lctx.mk_lambda(fvars, r);
8807 }
8808
8809 type_checker::type_checker(environment const &env, local_ctx const &lctx,
8810     bool safe_only)
8811     : m_st_owner(true),
8812      m_st(new state(env)),
8813      m_lctx(lctx),
8814      m_safe_only(safe_only),
8815      m_lparams(nullptr) {}
8816
8817 type_checker::type_checker(state &st, local_ctx const &lctx, bool safe_only)
8818     : m_st_owner(false),
8819      m_st(&st),

```

```

8820     m_lctx(lctx),
8821     m_safe_only(safe_only),
8822     m_lparams(nullptr) {}
8823
8824 type_checker::type_checker(type_checker &&src)
8825     : m_st_owner(src.m_st_owner),
8826       m_st(src.m_st),
8827       m_lctx(std::move(src.m_lctx)),
8828       m_safe_only(src.m_safe_only),
8829       m_lparams(src.m_lparams) {
8830     src.m_st_owner = false;
8831 }
8832
8833 type_checker::~~type_checker() {
8834     if (m_st_owner) delete m_st;
8835 }
8836
8837 extern "C" uint8 lean_kernel_is_def_eq(lean_object *env, lean_object *lctx,
8838                                       lean_object *a, lean_object *b) {
8839     return type_checker(environment(env), local_ctx(lctx))
8840         .is_def_eq(expr(a), expr(b));
8841 }
8842
8843 extern "C" lean_object *lean_kernel_wnhf(lean_object *env, lean_object *lctx,
8844                                         lean_object *a) {
8845     return type_checker(environment(env), local_ctx(lctx))
8846         .wnhf(expr(a))
8847         .steal();
8848 }
8849
8850 void initialize_type_checker() {
8851     g_dont_care = new expr(mk_const("dontcare"));
8852     mark_persistent(g_dont_care->raw());
8853     g_kernel_fresh = new name("_kernel_fresh");
8854     mark_persistent(g_kernel_fresh->raw());
8855     g_nat_zero = new expr(mk_constant(name{"Nat", "zero"}));
8856     mark_persistent(g_nat_zero->raw());
8857     g_nat_succ = new expr(mk_constant(name{"Nat", "succ"}));
8858     mark_persistent(g_nat_succ->raw());
8859     g_nat_add = new expr(mk_constant(name{"Nat", "add"}));
8860     mark_persistent(g_nat_add->raw());
8861     g_nat_sub = new expr(mk_constant(name{"Nat", "sub"}));
8862     mark_persistent(g_nat_sub->raw());
8863     g_nat_mul = new expr(mk_constant(name{"Nat", "mul"}));
8864     mark_persistent(g_nat_mul->raw());
8865     g_nat_div = new expr(mk_constant(name{"Nat", "div"}));
8866     mark_persistent(g_nat_div->raw());
8867     g_nat_mod = new expr(mk_constant(name{"Nat", "mod"}));
8868     mark_persistent(g_nat_mod->raw());
8869     g_nat_beq = new expr(mk_constant(name{"Nat", "beq"}));
8870     mark_persistent(g_nat_beq->raw());
8871     g_nat_ble = new expr(mk_constant(name{"Nat", "ble"}));
8872     mark_persistent(g_nat_ble->raw());
8873     g_string_mk = new expr(mk_constant(name{"String", "mk"}));
8874     mark_persistent(g_string_mk->raw());
8875     g_lean_reduce_bool = new expr(mk_constant(name{"Lean", "reduceBool"}));
8876     mark_persistent(g_lean_reduce_bool->raw());
8877     g_lean_reduce_nat = new expr(mk_constant(name{"Lean", "reduceNat"}));
8878     mark_persistent(g_lean_reduce_nat->raw());
8879     register_name_generator_prefix(*g_kernel_fresh);
8880 }
8881
8882 void finalize_type_checker() {
8883     delete g_dont_care;
8884     delete g_kernel_fresh;
8885     delete g_nat_succ;
8886     delete g_nat_zero;
8887     delete g_nat_add;
8888     delete g_nat_sub;
8889     delete g_nat_mul;

```

```
8890     delete g_nat_div;
8891     delete g_nat_mod;
8892     delete g_nat_beq;
8893     delete g_nat_ble;
8894     delete g_string_mk;
8895     delete g_lean_reduce_bool;
8896     delete g_lean_reduce_nat;
8897 }
8898 } // namespace lean
```