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

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

#include <memory>
#include <string>
#include <unordered_map>

#include "library/annotation.h"
#include "util/name_hash_map.h"

namespace lean {
static name *g_annotation = nullptr;

kvmap mk_annotation_kvmap(name const &k) {
    return set_name(kvmap(), *g_annotation, k);
}

typedef name_hash_map<kvmap> annotation_maps;
static annotation_maps *g_annotation_maps = nullptr;

void register_annotation(name const &kind) {
    lean_assert(g_annotation_maps->find(kind) == g_annotation_maps->end());
    g_annotation_maps->insert(mk_pair(kind, mk_annotation_kvmap(kind)));
}

optional<expr> is_annotation(expr const &e) {
    expr e2 = e;
    if (is_mdata(e2) && get_name(mdata_data(e2), *g_annotation))
        return some_expr(e2);
    else
        return none_expr();
}

name get_annotation_kind(expr const &e) {
    auto o = is_annotation(e);
    lean_assert(o);
    return *get_name(mdata_data(*o), *g_annotation);
}

bool is_annotation(expr const &e, name const &kind) {
    auto o = is_annotation(e);
    return o && get_annotation_kind(*o) == kind;
}

expr const &get_annotation_arg(expr const &e) {
    auto o = is_annotation(e);
    lean_assert(o);
    return mdata_expr(*o);
}

expr mk_annotation(name const &kind, expr const &e) {
    auto it = g_annotation_maps->find(kind);
    if (it != g_annotation_maps->end()) {
        expr r = mk_mdata(it->second, e);
        lean_assert(is_annotation(r));
        lean_assert(get_annotation_kind(r) == kind);
        return r;
    } else {
        throw exception(sstream()
                        << "unknown annotation kind '" << kind << "'");
    }
```

```
 70 }
 71
 72 bool is_nested_annotation(expr const &e, name const &kind) {
 73     expr const *it = &e;
 74     while (is_annotation(*it)) {
 75         if (get_annotation_kind(*it) == kind) return true;
 76         it = &get_annotation_arg(*it);
 77     }
 78     return false;
 79 }
 80
 81 expr const &get_nested_annotation_arg(expr const &e) {
 82     expr const *it = &e;
 83     while (is_annotation(*it)) it = &get_annotation_arg(*it);
 84     return *it;
 85 }
 86
 87 expr copy_annotations(expr const &from, expr const &to) {
 88     buffer<expr> trace;
 89     expr const *it = &from;
 90     while (is_annotation(*it)) {
 91         trace.push_back(*it);
 92         it = &get_annotation_arg(*it);
 93     }
 94     expr r = to;
 95     unsigned i = trace.size();
 96     while (i > 0) {
 97         --i;
 98         r = mk_annotation(get_annotation_kind(trace[i]), r);
 99     }
100     return r;
101 }
102
103 static name *g_have = nullptr;
104 static name *g_show = nullptr;
105 static name *g_suffices = nullptr;
106 static name *g_checkpoint = nullptr;
107
108 expr mk_have_annotation(expr const &e) { return mk_annotation(*g_have, e); }
109 expr mk_show_annotation(expr const &e) { return mk_annotation(*g_show, e); }
110 expr mk_suffices_annotation(expr const &e) {
111     return mk_annotation(*g_suffices, e);
112 }
113 expr mk_checkpoint_annotation(expr const &e) {
114     return mk_annotation(*g_checkpoint, e);
115 }
116 bool is_have_annotation(expr const &e) { return is_annotation(e, *g_have); }
117 bool is_show_annotation(expr const &e) { return is_annotation(e, *g_show); }
118 bool is_suffices_annotation(expr const &e) {
119     return is_annotation(e, *g_suffices);
120 }
121 bool is_checkpoint_annotation(expr const &e) {
122     return is_annotation(e, *g_checkpoint);
123 }
124
125 void initialize_annotation() {
126     g_annotation = new name("annotation");
127     mark_persistent(g_annotation->raw());
128     g_annotation_maps = new annotation_maps();
129     g_have = new name("have");
130     mark_persistent(g_have->raw());
131     g_show = new name("show");
132     mark_persistent(g_show->raw());
133     g_suffices = new name("suffices");
134     mark_persistent(g_suffices->raw());
135     g_checkpoint = new name("checkpoint");
136     mark_persistent(g_checkpoint->raw());
137
138     register_annotation(*g_have);
139     register_annotation(*g_show);
```

```
140     register_annotation(*g_suffices);
141     register_annotation(*g_checkpoint);
142 }
143
144 void finalize_annotation() {
145     delete g_checkpoint;
146     delete g_show;
147     delete g_have;
148     delete g_suffices;
149     delete g_annotation;
150 }
151 }  // namespace lean
152 //::::::::::::::
153 // aux_recursors.cpp
154 //::::::::::::::
155 /*
156 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
157 Released under Apache 2.0 license as described in the file LICENSE.
158
159 Author: Leonardo de Moura
160 */
161 #include "library/aux_recursors.h"
162
163 namespace lean {
164 extern "C" object *lean_mark_aux_recursor(object *env, object *n);
165 extern "C" object *lean_mark_no_confusion(object *env, object *n);
166 extern "C" uint8 lean_is_aux_recursor(object *env, object *n);
167 extern "C" uint8 lean_is_no_confusion(object *env, object *n);
168
169 environment add_aux_recursor(environment const &env, name const &r) {
170     return environment(
171         lean_mark_aux_recursor(env.to_obj_arg(), r.to_obj_arg()));
172 }
173
174 environment add_no_confusion(environment const &env, name const &r) {
175     return environment(
176         lean_mark_no_confusion(env.to_obj_arg(), r.to_obj_arg()));
177 }
178
179 bool is_aux_recursor(environment const &env, name const &r) {
180     return lean_is_aux_recursor(env.to_obj_arg(), r.to_obj_arg());
181 }
182
183 bool is_no_confusion(environment const &env, name const &r) {
184     return lean_is_no_confusion(env.to_obj_arg(), r.to_obj_arg());
185 }
186 }  // namespace lean
187 // ::::::::::::::
188 // bin_app.cpp
189 // ::::::::::::::
190 /*
191 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
192 Released under Apache 2.0 license as described in the file LICENSE.
193
194 Author: Leonardo de Moura
195 */
196 #include "library/bin_app.h"
197
198 namespace lean {
199 bool is_bin_app(expr const &t, expr const &f) {
200     return is_app(t) && is_app(app_fn(t)) && app_fn(app_fn(t)) == f;
201 }
202
203 bool is_bin_app(expr const &t, expr const &f, expr &lhs, expr &rhs) {
204     if (is_bin_app(t, f)) {
205         lhs = app_arg(app_fn(t));
206         rhs = app_arg(t);
207         return true;
208     } else {
209         return false;
```

```
210     }
211 }
212
213 expr mk_bin_rop(expr const &op, expr const &unit, unsigned num_args,
214                 expr const *args) {
215     if (num_args == 0) {
216         return unit;
217     } else {
218         expr r = args[num_args - 1];
219         unsigned i = num_args - 1;
220         while (i > 0) {
221             --i;
222             r = mk_app(op, args[i], r);
223         }
224         return r;
225     }
226 }
227 expr mk_bin_rop(expr const &op, expr const &unit,
228                 std::initializer_list<expr> const &l) {
229     return mk_bin_rop(op, unit, l.size(), l.begin());
230 }
231
232 expr mk_bin_lop(expr const &op, expr const &unit, unsigned num_args,
233                 expr const *args) {
234     if (num_args == 0) {
235         return unit;
236     } else {
237         expr r = args[0];
238         for (unsigned i = 1; i < num_args; i++) {
239             r = mk_app(op, r, args[i]);
240         }
241         return r;
242     }
243 }
244 expr mk_bin_lop(expr const &op, expr const &unit,
245                 std::initializer_list<expr> const &l) {
246     return mk_bin_lop(op, unit, l.size(), l.begin());
247 }
248 }  // namespace lean
249 // ::::::::::::::
250 // class.cpp
251 // ::::::::::::::
252 /*
253 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
254 Released under Apache 2.0 license as described in the file LICENSE.
255
256 Author: Leonardo de Moura
257 */
258 #include <string>
259
260 #include "library/class.h"
261 #include "util/io.h"
262
263 namespace lean {
264 extern "C" uint8 lean_is_class(object *env, object *n);
265 extern "C" uint8 lean_is_instance(object *env, object *n);
266 extern "C" uint8 lean_is_out_param(object *e);
267 extern "C" uint8 lean_has_out_params(object *env, object *n);
268
269 bool is_class_out_param(expr const &e) {
270     return lean_is_out_param(e.to_obj_arg());
271 }
272 bool has_class_out_params(environment const &env, name const &c) {
273     return lean_has_out_params(env.to_obj_arg(), c.to_obj_arg());
274 }
275 bool is_class(environment const &env, name const &c) {
276     return lean_is_class(env.to_obj_arg(), c.to_obj_arg());
277 }
278 bool is_instance(environment const &env, name const &i) {
279     return lean_is_instance(env.to_obj_arg(), i.to_obj_arg());
```

```
280 }
281
282 static name *g_anonymous_inst_name_prefix = nullptr;
283
284 name const &get_anonymous_instance_prefix() {
285     return *g_anonymous_inst_name_prefix;
286 }
287
288 name mk_anonymous_inst_name(unsigned idx) {
289     return g_anonymous_inst_name_prefix->append_after(idx);
290 }
291
292 bool is_anonymous_inst_name(name const &n) {
293     // remove mangled macro scopes
294     auto n2 = n.get_root();
295     if (!n2.is_string()) return false;
296     return strncmp(n2.get_string().data(),
297                    g_anonymous_inst_name_prefix->get_string().data(),
298                    strlen(g_anonymous_inst_name_prefix->get_string().data())) ==
299            0;
300 }
301
302 void initialize_class() {
303     g_anonymous_inst_name_prefix = new name("_inst");
304     mark_persistent(g_anonymous_inst_name_prefix->raw());
305 }
306
307 void finalize_class() { delete g_anonymous_inst_name_prefix; }
308 }  // namespace lean
309 // ::::::::::::::
310 // constants.cpp
311 // ::::::::::::::
312 // Copyright (c) 2015 Microsoft Corporation. All rights reserved.
313 // Released under Apache 2.0 license as described in the file LICENSE.
314 // DO NOT EDIT, automatically generated file, generator
315 // scripts/gen_constants_cpp.py
316 #include "util/name.h"
317 namespace lean {
318 name const *g_absurd = nullptr;
319 name const *g_and = nullptr;
320 name const *g_and_left = nullptr;
321 name const *g_and_right = nullptr;
322 name const *g_and_intro = nullptr;
323 name const *g_and_rec = nullptr;
324 name const *g_and_cases_on = nullptr;
325 name const *g_array = nullptr;
326 name const *g_array_sz = nullptr;
327 name const *g_array_data = nullptr;
328 name const *g_auto_param = nullptr;
329 name const *g_bit0 = nullptr;
330 name const *g_bit1 = nullptr;
331 name const *g_has_of_nat_of_nat = nullptr;
332 name const *g_byte_array = nullptr;
333 name const *g_bool = nullptr;
334 name const *g_bool_false = nullptr;
335 name const *g_bool_true = nullptr;
336 name const *g_bool_cases_on = nullptr;
337 name const *g_cast = nullptr;
338 name const *g_char = nullptr;
339 name const *g_congr_arg = nullptr;
340 name const *g_decidable = nullptr;
341 name const *g_decidable_is_true = nullptr;
342 name const *g_decidable_is_false = nullptr;
343 name const *g_decidable_decide = nullptr;
344 name const *g_empty = nullptr;
345 name const *g_empty_rec = nullptr;
346 name const *g_empty_cases_on = nullptr;
347 name const *g_exists = nullptr;
348 name const *g_eq = nullptr;
349 name const *g_eq_cases_on = nullptr;
```

```
name const *g_eq_rec_on = nullptr;
name const *g_eq_rec = nullptr;
name const *g_eq_ndrec = nullptr;
name const *g_eq_refl = nullptr;
name const *g_eq_subst = nullptr;
name const *g_eq_symm = nullptr;
name const *g_eq_trans = nullptr;
name const *g_float = nullptr;
name const *g_float_array = nullptr;
name const *g_false = nullptr;
name const *g_false_rec = nullptr;
name const *g_false_cases_on = nullptr;
name const *g_has_add_add = nullptr;
name const *g_has_neg_neg = nullptr;
name const *g_has_one_one = nullptr;
name const *g_has_zero_zero = nullptr;
name const *g_heq = nullptr;
name const *g_heq_refl = nullptr;
name const *g_iff = nullptr;
name const *g_iff_refl = nullptr;
name const *g_int = nullptr;
name const *g_int_nat_abs = nullptr;
name const *g_int_dec_lt = nullptr;
name const *g_int_of_nat = nullptr;
name const *g_inline = nullptr;
name const *g_io = nullptr;
name const *g_ite = nullptr;
name const *g_lc_proof = nullptr;
name const *g_lc_unreachable = nullptr;
name const *g_list = nullptr;
name const *g_mut_quot = nullptr;
name const *g_nat = nullptr;
name const *g_nat_succ = nullptr;
name const *g_nat_zero = nullptr;
name const *g_nat_has_zero = nullptr;
name const *g_nat_has_one = nullptr;
name const *g_nat_has_add = nullptr;
name const *g_nat_add = nullptr;
name const *g_nat_dec_eq = nullptr;
name const *g_nat_sub = nullptr;
name const *g_ne = nullptr;
name const *g_not = nullptr;
name const *g_opt_param = nullptr;
name const *g_or = nullptr;
name const *g_panic = nullptr;
name const *g_punit = nullptr;
name const *g_punit_unit = nullptr;
name const *g_pprod = nullptr;
name const *g_pprod_mk = nullptr;
name const *g_pprod_fst = nullptr;
name const *g_pprod_snd = nullptr;
name const *g_propext = nullptr;
name const *g_quot_mk = nullptr;
name const *g_quot_lift = nullptr;
name const *g_sorry_ax = nullptr;
name const *g_string = nullptr;
name const *g_string_data = nullptr;
name const *g_subsingleton_elim = nullptr;
name const *g_task = nullptr;
name const *g_thunk = nullptr;
name const *g_thunk_mk = nullptr;
name const *g_thunk_get = nullptr;
name const *g_true = nullptr;
name const *g_true_intro = nullptr;
name const *g_unit = nullptr;
name const *g_unit_unit = nullptr;
name const *g_uint8 = nullptr;
name const *g_uint16 = nullptr;
name const *g_uint32 = nullptr;
name const *g_uint64 = nullptr;
```

```
name *const *g_usize = nullptr;
void initialize_constants() {
    g_absurd = new name{"absurd"};
    mark_persistent(g_absurd->raw());
    g_and = new name{"And"};
    mark_persistent(g_and->raw());
    g_and_left = new name{"And", "left"};
    mark_persistent(g_and_left->raw());
    g_and_right = new name{"And", "right"};
    mark_persistent(g_and_right->raw());
    g_and_intro = new name{"And", "intro"};
    mark_persistent(g_and_intro->raw());
    g_and_rec = new name{"And", "rec"};
    mark_persistent(g_and_rec->raw());
    g_and_cases_on = new name{"And", "casesOn"};
    mark_persistent(g_and_cases_on->raw());
    g_array = new name{"Array"};
    mark_persistent(g_array->raw());
    g_array_sz = new name{"Array", "sz"};
    mark_persistent(g_array_sz->raw());
    g_array_data = new name{"Array", "data"};
    mark_persistent(g_array_data->raw());
    g_auto_param = new name{"autoParam"};
    mark_persistent(g_auto_param->raw());
    g_bit0 = new name{"bit0"};
    mark_persistent(g_bit0->raw());
    g_bit1 = new name{"bit1"};
    mark_persistent(g_bit1->raw());
    g_has_of_nat_of_nat = new name{"HasOfNat", "ofNat"};
    mark_persistent(g_has_of_nat_of_nat->raw());
    g_byte_array = new name{"ByteArray"};
    mark_persistent(g_byte_array->raw());
    g_bool = new name{"Bool"};
    mark_persistent(g_bool->raw());
    g_bool_false = new name{"Bool", "false"};
    mark_persistent(g_bool_false->raw());
    g_bool_true = new name{"Bool", "true"};
    mark_persistent(g_bool_true->raw());
    g_bool_cases_on = new name{"Bool", "casesOn"};
    mark_persistent(g_bool_cases_on->raw());
    g_cast = new name{"cast"};
    mark_persistent(g_cast->raw());
    g_char = new name{"Char"};
    mark_persistent(g_char->raw());
    g_congr_arg = new name{"congrArg"};
    mark_persistent(g_congr_arg->raw());
    g_decidable = new name{"Decidable"};
    mark_persistent(g_decidable->raw());
    g_decidable_is_true = new name{"Decidable", "isTrue"};
    mark_persistent(g_decidable_is_true->raw());
    g_decidable_is_false = new name{"Decidable", "isFalse"};
    mark_persistent(g_decidable_is_false->raw());
    g_decidable_decide = new name{"Decidable", "decide"};
    mark_persistent(g_decidable_decide->raw());
    g_empty = new name{"Empty"};
    mark_persistent(g_empty->raw());
    g_empty_rec = new name{"Empty", "rec"};
    mark_persistent(g_empty_rec->raw());
    g_empty_cases_on = new name{"Empty", "casesOn"};
    mark_persistent(g_empty_cases_on->raw());
    g_exists = new name{"Exists"};
    mark_persistent(g_exists->raw());
    g_eq = new name{"Eq"};
    mark_persistent(g_eq->raw());
    g_eq_cases_on = new name{"Eq", "casesOn"};
    mark_persistent(g_eq_cases_on->raw());
    g_eq_rec_on = new name{"Eq", "recOn"};
    mark_persistent(g_eq_rec_on->raw());
    g_eq_rec = new name{"Eq", "rec"};
    mark_persistent(g_eq_rec->raw());
```

```
490     g_eq_ndrec = new name{"Eq", "ndrec"};
491     mark_persistent(g_eq_ndrec->raw());
492     g_eq_refl = new name{"Eq", "refl"};
493     mark_persistent(g_eq_refl->raw());
494     g_eq_subst = new name{"Eq", "subst"};
495     mark_persistent(g_eq_subst->raw());
496     g_eq_symm = new name{"Eq", "symm"};
497     mark_persistent(g_eq_symm->raw());
498     g_eq_trans = new name{"Eq", "trans"};
499     mark_persistent(g_eq_trans->raw());
500     g_float = new name{"Float"};
501     mark_persistent(g_float->raw());
502     g_float_array = new name{"FloatArray"};
503     mark_persistent(g_float_array->raw());
504     g_false = new name{"False"};
505     mark_persistent(g_false->raw());
506     g_false_rec = new name{"False", "rec"};
507     mark_persistent(g_false_rec->raw());
508     g_false_cases_on = new name{"False", "casesOn"};
509     mark_persistent(g_false_cases_on->raw());
510     g_has_add_add = new name{"HasAdd", "add"};
511     mark_persistent(g_has_add_add->raw());
512     g_has_neg_neg = new name{"HasNeg", "neg"};
513     mark_persistent(g_has_neg_neg->raw());
514     g_has_one_one = new name{"HasOne", "one"};
515     mark_persistent(g_has_one_one->raw());
516     g_has_zero_zero = new name{"HasZero", "zero"};
517     mark_persistent(g_has_zero_zero->raw());
518     g_heq = new name{"HEq"};
519     mark_persistent(g_heq->raw());
520     g_heq_refl = new name{"HEq", "refl"};
521     mark_persistent(g_heq_refl->raw());
522     g_iff = new name{"Iff"};
523     mark_persistent(g_iff->raw());
524     g_iff_refl = new name{"Iff", "refl"};
525     mark_persistent(g_iff_refl->raw());
526     g_int = new name{"Int"};
527     mark_persistent(g_int->raw());
528     g_int_nat_abs = new name{"Int", "natAbs"};
529     mark_persistent(g_int_nat_abs->raw());
530     g_int_dec_lt = new name{"Int", "decLt"};
531     mark_persistent(g_int_dec_lt->raw());
532     g_int_of_nat = new name{"Int", "ofNat"};
533     mark_persistent(g_int_of_nat->raw());
534     g_inline = new name{"inline"};
535     mark_persistent(g_inline->raw());
536     g_io = new name{"IO"};
537     mark_persistent(g_io->raw());
538     g_ite = new name{"ite"};
539     mark_persistent(g_ite->raw());
540     g_lc_proof = new name{"lcProof"};
541     mark_persistent(g_lc_proof->raw());
542     g_lc_unreachable = new name{"lcUnreachable"};
543     mark_persistent(g_lc_unreachable->raw());
544     g_list = new name{"List"};
545     mark_persistent(g_list->raw());
546     g_mut_quot = new name{"MutQuot"};
547     mark_persistent(g_mut_quot->raw());
548     g_nat = new name{"Nat"};
549     mark_persistent(g_nat->raw());
550     g_nat_succ = new name{"Nat", "succ"};
551     mark_persistent(g_nat_succ->raw());
552     g_nat_zero = new name{"Nat", "zero"};
553     mark_persistent(g_nat_zero->raw());
554     g_nat_has_zero = new name{"Nat", "HasZero"};
555     mark_persistent(g_nat_has_zero->raw());
556     g_nat_has_one = new name{"Nat", "HasOne"};
557     mark_persistent(g_nat_has_one->raw());
558     g_nat_has_add = new name{"Nat", "HasAdd"};
559     mark_persistent(g_nat_has_add->raw());
```

```
560     g_nat_add = new name{"Nat", "add"};
561     mark_persistent(g_nat_add->raw());
562     g_nat_dec_eq = new name{"Nat", "decEq"};
563     mark_persistent(g_nat_dec_eq->raw());
564     g_nat_sub = new name{"Nat", "sub"};
565     mark_persistent(g_nat_sub->raw());
566     g_ne = new name{"ne"};
567     mark_persistent(g_ne->raw());
568     g_not = new name{"Not"};
569     mark_persistent(g_not->raw());
570     g_opt_param = new name{"optParam"};
571     mark_persistent(g_opt_param->raw());
572     g_or = new name{"Or"};
573     mark_persistent(g_or->raw());
574     g_panic = new name{"panic"};
575     mark_persistent(g_panic->raw());
576     g_punit = new name{"PUnit"};
577     mark_persistent(g_punit->raw());
578     g_punit_unit = new name{"PUnit", "unit"};
579     mark_persistent(g_punit_unit->raw());
580     g_pprod = new name{"PProd"};
581     mark_persistent(g_pprod->raw());
582     g_pprod_mk = new name{"PProd", "mk"};
583     mark_persistent(g_pprod_mk->raw());
584     g_pprod_fst = new name{"PProd", "fst"};
585     mark_persistent(g_pprod_fst->raw());
586     g_pprod_snd = new name{"PProd", "snd"};
587     mark_persistent(g_pprod_snd->raw());
588     g_propext = new name{"propext"};
589     mark_persistent(g_propext->raw());
590     g_quot_mk = new name{"Quot", "mk"};
591     mark_persistent(g_quot_mk->raw());
592     g_quot_lift = new name{"Quot", "lift"};
593     mark_persistent(g_quot_lift->raw());
594     g_sorry_ax = new name{"sorryAx"};
595     mark_persistent(g_sorry_ax->raw());
596     g_string = new name{"String"};
597     mark_persistent(g_string->raw());
598     g_string_data = new name{"String", "data"};
599     mark_persistent(g_string_data->raw());
600     g_subsingleton_elim = new name{"Subsingleton", "elim"};
601     mark_persistent(g_subsingleton_elim->raw());
602     g_task = new name{"Task"};
603     mark_persistent(g_task->raw());
604     g_thunk = new name{"Thunk"};
605     mark_persistent(g_thunk->raw());
606     g_thunk_mk = new name{"Thunk", "mk"};
607     mark_persistent(g_thunk_mk->raw());
608     g_thunk_get = new name{"Thunk", "get"};
609     mark_persistent(g_thunk_get->raw());
610     g_true = new name{"True"};
611     mark_persistent(g_true->raw());
612     g_true_intro = new name{"True", "intro"};
613     mark_persistent(g_true_intro->raw());
614     g_unit = new name{"Unit"};
615     mark_persistent(g_unit->raw());
616     g_unit_unit = new name{"Unit", "unit"};
617     mark_persistent(g_unit_unit->raw());
618     g_uint8 = new name{"UInt8"};
619     mark_persistent(g_uint8->raw());
620     g_uint16 = new name{"UInt16"};
621     mark_persistent(g_uint16->raw());
622     g_uint32 = new name{"UInt32"};
623     mark_persistent(g_uint32->raw());
624     g_uint64 = new name{"UInt64"};
625     mark_persistent(g_uint64->raw());
626     g_usize = new name{"USize"};
627     mark_persistent(g_usize->raw());
628 }
629 void finalize_constants() {
```

```
630     delete g_absurd;
631     delete g_and;
632     delete g_and_left;
633     delete g_and_right;
634     delete g_and_intro;
635     delete g_and_rec;
636     delete g_and_cases_on;
637     delete g_array;
638     delete g_array_sz;
639     delete g_array_data;
640     delete g_auto_param;
641     delete g_bit0;
642     delete g_bit1;
643     delete g_has_of_nat_of_nat;
644     delete g_byte_array;
645     delete g_bool;
646     delete g_bool_false;
647     delete g_bool_true;
648     delete g_bool_cases_on;
649     delete g_cast;
650     delete g_char;
651     delete g_congr_arg;
652     delete g_decidable;
653     delete g_decidable_is_true;
654     delete g_decidable_is_false;
655     delete g_decidable_decide;
656     delete g_empty;
657     delete g_empty_rec;
658     delete g_empty_cases_on;
659     delete g_exists;
660     delete g_eq;
661     delete g_eq_cases_on;
662     delete g_eq_rec_on;
663     delete g_eq_rec;
664     delete g_eq_ndrec;
665     delete g_eq_refl;
666     delete g_eq_subst;
667     delete g_eq_symm;
668     delete g_eq_trans;
669     delete g_float;
670     delete g_float_array;
671     delete g_false;
672     delete g_false_rec;
673     delete g_false_cases_on;
674     delete g_has_add_add;
675     delete g_has_neg_neg;
676     delete g_has_one_one;
677     delete g_has_zero_zero;
678     delete g_heq;
679     delete g_heq_refl;
680     delete g_iff;
681     delete g_iff_refl;
682     delete g_int;
683     delete g_int_nat_abs;
684     delete g_int_dec_lt;
685     delete g_int_of_nat;
686     delete g_inline;
687     delete g_io;
688     delete g_ite;
689     delete g_lc_proof;
690     delete g_lc_unreachable;
691     delete g_list;
692     delete g_mut_quot;
693     delete g_nat;
694     delete g_nat_succ;
695     delete g_nat_zero;
696     delete g_nat_has_zero;
697     delete g_nat_has_one;
698     delete g_nat_has_add;
699     delete g_nat_add;
```

```
700     delete g_nat_dec_eq;
701     delete g_nat_sub;
702     delete g_ne;
703     delete g_not;
704     delete g_opt_param;
705     delete g_or;
706     delete g_panic;
707     delete g_punit;
708     delete g_punit_unit;
709     delete g_pprod;
710     delete g_pprod_mk;
711     delete g_pprod_fst;
712     delete g_pprod_snd;
713     delete g_propext;
714     delete g_quot_mk;
715     delete g_quot_lift;
716     delete g_sorry_ax;
717     delete g_string;
718     delete g_string_data;
719     delete g_subsingleton_elim;
720     delete g_task;
721     delete g_thunk;
722     delete g_thunk_mk;
723     delete g_thunk_get;
724     delete g_true;
725     delete g_true_intro;
726     delete g_unit;
727     delete g_unit_unit;
728     delete g_uint8;
729     delete g_uint16;
730     delete g_uint32;
731     delete g_uint64;
732     delete g_usize;
733 }
734 name const &get_absurd_name() { return *g_absurd; }
735 name const &get_and_name() { return *g_and; }
736 name const &get_and_left_name() { return *g_and_left; }
737 name const &get_and_right_name() { return *g_and_right; }
738 name const &get_and_intro_name() { return *g_and_intro; }
739 name const &get_and_rec_name() { return *g_and_rec; }
740 name const &get_and_cases_on_name() { return *g_and_cases_on; }
741 name const &get_array_name() { return *g_array; }
742 name const &get_array_sz_name() { return *g_array_sz; }
743 name const &get_array_data_name() { return *g_array_data; }
744 name const &get_auto_param_name() { return *g_auto_param; }
745 name const &get_bit0_name() { return *g_bit0; }
746 name const &get_bit1_name() { return *g_bit1; }
747 name const &get_has_of_nat_of_nat_name() { return *g_has_of_nat_of_nat; }
748 name const &get_byte_array_name() { return *g_byte_array; }
749 name const &get_bool_name() { return *g_bool; }
750 name const &get_bool_false_name() { return *g_bool_false; }
751 name const &get_bool_true_name() { return *g_bool_true; }
752 name const &get_bool_cases_on_name() { return *g_bool_cases_on; }
753 name const &get_cast_name() { return *g_cast; }
754 name const &get_char_name() { return *g_char; }
755 name const &get_congr_arg_name() { return *g_congr_arg; }
756 name const &get_decidable_name() { return *g_decidable; }
757 name const &get_decidable_is_true_name() { return *g_decidable_is_true; }
758 name const &get_decidable_is_false_name() { return *g_decidable_is_false; }
759 name const &get_decidable_decide_name() { return *g_decidable_decide; }
760 name const &get_empty_name() { return *g_empty; }
761 name const &get_empty_rec_name() { return *g_empty_rec; }
762 name const &get_empty_cases_on_name() { return *g_empty_cases_on; }
763 name const &get_exists_name() { return *g_exists; }
764 name const &get_eq_name() { return *g_eq; }
765 name const &get_eq_cases_on_name() { return *g_eq_cases_on; }
766 name const &get_eq_rec_on_name() { return *g_eq_rec_on; }
767 name const &get_eq_rec_name() { return *g_eq_rec; }
768 name const &get_eq_ndrec_name() { return *g_eq_ndrec; }
769 name const &get_eq_refl_name() { return *g_eq_refl; }
```

```
name const &get_eq_subst_name() { return *g_eq_subst; }
name const &get_eq_symm_name() { return *g_eq_symm; }
name const &get_eq_trans_name() { return *g_eq_trans; }
name const &get_float_name() { return *g_float; }
name const &get_float_array_name() { return *g_float_array; }
name const &get_false_name() { return *g_false; }
name const &get_false_rec_name() { return *g_false_rec; }
name const &get_false_cases_on_name() { return *g_false_cases_on; }
name const &get_has_add_add_name() { return *g_has_add_add; }
name const &get_has_neg_neg_name() { return *g_has_neg_neg; }
name const &get_has_one_one_name() { return *g_has_one_one; }
name const &get_has_zero_zero_name() { return *g_has_zero_zero; }
name const &get_heq_name() { return *g_heq; }
name const &get_heq_refl_name() { return *g_heq_refl; }
name const &get_iff_name() { return *g_iff; }
name const &get_iff_refl_name() { return *g_iff_refl; }
name const &get_int_name() { return *g_int; }
name const &get_int_nat_abs_name() { return *g_int_nat_abs; }
name const &get_int_dec_lt_name() { return *g_int_dec_lt; }
name const &get_int_of_nat_name() { return *g_int_of_nat; }
name const &get_inline_name() { return *g_inline; }
name const &get_io_name() { return *g_io; }
name const &get_ite_name() { return *g_ite; }
name const &get_lc_proof_name() { return *g_lc_proof; }
name const &get_lc_unreachable_name() { return *g_lc_unreachable; }
name const &get_list_name() { return *g_list; }
name const &get_mut_quot_name() { return *g_mut_quot; }
name const &get_nat_name() { return *g_nat; }
name const &get_nat_succ_name() { return *g_nat_succ; }
name const &get_nat_zero_name() { return *g_nat_zero; }
name const &get_nat_has_zero_name() { return *g_nat_has_zero; }
name const &get_nat_has_one_name() { return *g_nat_has_one; }
name const &get_nat_has_add_name() { return *g_nat_has_add; }
name const &get_nat_add_name() { return *g_nat_add; }
name const &get_nat_dec_eq_name() { return *g_nat_dec_eq; }
name const &get_nat_sub_name() { return *g_nat_sub; }
name const &get_ne_name() { return *g_ne; }
name const &get_not_name() { return *g_not; }
name const &get_opt_param_name() { return *g_opt_param; }
name const &get_or_name() { return *g_or; }
name const &get_panic_name() { return *g_panic; }
name const &get_punit_name() { return *g_punit; }
name const &get_punit_unit_name() { return *g_punit_unit; }
name const &get_pprod_name() { return *g_pprod; }
name const &get_pprod_mk_name() { return *g_pprod_mk; }
name const &get_pprod_fst_name() { return *g_pprod_fst; }
name const &get_pprod_snd_name() { return *g_pprod_snd; }
name const &get_propext_name() { return *g_propext; }
name const &get_quot_mk_name() { return *g_quot_mk; }
name const &get_quot_lift_name() { return *g_quot_lift; }
name const &get_sorry_ax_name() { return *g_sorry_ax; }
name const &get_string_name() { return *g_string; }
name const &get_string_data_name() { return *g_string_data; }
name const &get_subsingleton_elim_name() { return *g_subsingleton_elim; }
name const &get_task_name() { return *g_task; }
name const &get_thunk_name() { return *g_thunk; }
name const &get_thunk_mk_name() { return *g_thunk_mk; }
name const &get_thunk_get_name() { return *g_thunk_get; }
name const &get_true_name() { return *g_true; }
name const &get_true_intro_name() { return *g_true_intro; }
name const &get_unit_name() { return *g_unit; }
name const &get_unit_unit_name() { return *g_unit_unit; }
name const &get_uint8_name() { return *g_uint8; }
name const &get_uint16_name() { return *g_uint16; }
name const &get_uint32_name() { return *g_uint32; }
name const &get_uint64_name() { return *g_uint64; }
name const &get_usize_name() { return *g_usize; }
}  // namespace lean
// ::::::::::::::
// expr_lt.cpp
```

```
840 // ::::::::::::::
841 /*
842 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
843 Released under Apache 2.0 license as described in the file LICENSE.
844
845 Author: Leonardo de Moura
846 */
847 #include "kernel/expr.h"
848 #include "library/expr_lt.h"
849
850 namespace lean {
851 bool is_lt(expr const &a, expr const &b, bool use_hash, local_ctx const *lctx) {
852     if (is_eqp(a, b)) return false;
853     if (a.kind() != b.kind()) return a.kind() < b.kind();
854     if (use_hash) {
855         if (hash(a) < hash(b)) return true;
856         if (hash(a) > hash(b)) return false;
857     }
858     if (a == b) return false;
859     switch (a.kind()) {
860         case expr_kind::Lit:
861             return lit_value(a) < lit_value(b);
862         case expr_kind::BVar:
863             return bvar_idx(a) < bvar_idx(b);
864         case expr_kind::MData:
865             if (mdata_expr(a) != mdata_expr(b))
866                 return is_lt(mdata_expr(a), mdata_expr(b), use_hash, lctx);
867             else
868                 return mdata_data(a) < mdata_data(b);
869         case expr_kind::Proj:
870             if (proj_expr(a) != proj_expr(b))
871                 return is_lt(proj_expr(a), proj_expr(b), use_hash, lctx);
872             else if (proj_sname(a) != proj_sname(b))
873                 return proj_sname(a) < proj_sname(b);
874             else
875                 return proj_idx(a) < proj_idx(b);
876         case expr_kind::Const:
877             if (const_name(a) != const_name(b))
878                 return const_name(a) < const_name(b);
879             else
880                 return is_lt(const_levels(a), const_levels(b), use_hash);
881         case expr_kind::App:
882             if (app_fn(a) != app_fn(b))
883                 return is_lt(app_fn(a), app_fn(b), use_hash, lctx);
884             else
885                 return is_lt(app_arg(a), app_arg(b), use_hash, lctx);
886         case expr_kind::Lambda:
887         case expr_kind::Pi:
888             if (binding_domain(a) != binding_domain(b))
889                 return is_lt(binding_domain(a), binding_domain(b), use_hash,
890                              lctx);
891             else
892                 return is_lt(binding_body(a), binding_body(b), use_hash, lctx);
893         case expr_kind::Let:
894             if (let_type(a) != let_type(b))
895                 return is_lt(let_type(a), let_type(b), use_hash, lctx);
896             else if (let_value(a) != let_value(b))
897                 return is_lt(let_value(a), let_value(b), use_hash, lctx);
898             else
899                 return is_lt(let_body(a), let_body(b), use_hash, lctx);
900         case expr_kind::Sort:
901             return is_lt(sort_level(a), sort_level(b), use_hash);
902         case expr_kind::FVar:
903             if (lctx) {
904                 if (auto d1 = lctx->find_local_decl(a))
905                     if (auto d2 = lctx->find_local_decl(b))
906                         return d1->get_idx() < d2->get_idx();
907             }
908             return fvar_name(a) < fvar_name(b);
909         case expr_kind::MVar:
```

```
910             return mvar_name(a) < mvar_name(b);
911     }
912     lean_unreachable();  // LCOV_EXCL_LINE
913 }
914
915 bool is_lt_no_level_params(level const &a, level const &b) {
916     if (is_eqp(a, b)) return false;
917     if (kind(a) != kind(b)) {
918         if (kind(a) == level_kind::Param || kind(b) == level_kind::Param)
919             return false;
920         return kind(a) < kind(b);
921     }
922     switch (kind(a)) {
923         case level_kind::Zero:
924             lean_unreachable();  // LCOV_EXCL_LINE
925         case level_kind::Param:
926             return false;
927         case level_kind::MVar:
928             return mvar_id(a) < mvar_id(b);
929         case level_kind::Max:
930             if (is_lt_no_level_params(max_lhs(a), max_lhs(b)))
931                 return true;
932             else if (is_lt_no_level_params(max_lhs(b), max_lhs(a)))
933                 return false;
934             else
935                 return is_lt_no_level_params(max_rhs(a), max_rhs(b));
936         case level_kind::IMax:
937             if (is_lt_no_level_params(imax_lhs(a), imax_lhs(b)))
938                 return true;
939             else if (is_lt_no_level_params(imax_lhs(b), imax_lhs(a)))
940                 return false;
941             else
942                 return is_lt_no_level_params(imax_rhs(a), imax_rhs(b));
943         case level_kind::Succ:
944             return is_lt_no_level_params(succ_of(a), succ_of(b));
945     }
946     lean_unreachable();
947 }
948
949 bool is_lt_no_level_params(levels const &as, levels const &bs) {
950     if (is_nil(as))
951         return !is_nil(bs);
952     else if (is_nil(bs))
953         return false;
954     else if (is_lt_no_level_params(car(as), car(bs)))
955         return true;
956     else if (is_lt_no_level_params(car(bs), car(as)))
957         return false;
958     else
959         return is_lt_no_level_params(cdr(as), cdr(bs));
960 }
961
962 bool is_lt_no_level_params(expr const &a, expr const &b) {
963     if (is_eqp(a, b)) return false;
964     if (a.kind() != b.kind()) return a.kind() < b.kind();
965     switch (a.kind()) {
966         case expr_kind::Lit:
967             return lit_value(a) < lit_value(b);
968         case expr_kind::BVar:
969             return bvar_idx(a) < bvar_idx(b);
970         case expr_kind::MData:
971             if (mdata_expr(a) != mdata_expr(b))
972                 return is_lt_no_level_params(mdata_expr(a), mdata_expr(b));
973             else
974                 return mdata_data(a) < mdata_data(b);
975         case expr_kind::Proj:
976             if (proj_expr(a) != proj_expr(b))
977                 return is_lt_no_level_params(proj_expr(a), proj_expr(b));
978             else if (proj_sname(a) != proj_sname(b))
979                 return proj_sname(a) < proj_sname(b);
```

```
 980             else
 981                 return proj_idx(a) < proj_idx(b);
 982         case expr_kind::Const:
 983             if (const_name(a) != const_name(b))
 984                 return const_name(a) < const_name(b);
 985             else
 986                 return is_lt_no_level_params(const_levels(a), const_levels(b));
 987         case expr_kind::App:
 988             if (is_lt_no_level_params(app_fn(a), app_fn(b)))
 989                 return true;
 990             else if (is_lt_no_level_params(app_fn(b), app_fn(a)))
 991                 return false;
 992             else
 993                 return is_lt_no_level_params(app_arg(a), app_arg(b));
 994         case expr_kind::Lambda:
 995         case expr_kind::Pi:
 996             if (is_lt_no_level_params(binding_domain(a), binding_domain(b)))
 997                 return true;
 998             else if (is_lt_no_level_params(binding_domain(b),
 999                                            binding_domain(a)))
1000                 return false;
1001             else
1002                 return is_lt_no_level_params(binding_body(a), binding_body(b));
1003         case expr_kind::Let:
1004             if (is_lt_no_level_params(let_type(a), let_type(b)))
1005                 return true;
1006             else if (is_lt_no_level_params(let_type(b), let_type(a)))
1007                 return false;
1008             else if (is_lt_no_level_params(let_value(a), let_value(b)))
1009                 return true;
1010             else if (is_lt_no_level_params(let_value(b), let_value(a)))
1011                 return false;
1012             else
1013                 return is_lt_no_level_params(let_body(a), let_body(b));
1014         case expr_kind::Sort:
1015             return is_lt_no_level_params(sort_level(a), sort_level(b));
1016         case expr_kind::FVar:
1017             return fvar_name(a) < fvar_name(b);
1018         case expr_kind::MVar:
1019             return mvar_name(a) < mvar_name(b);
1020     }
1021     lean_unreachable();
1022 }
1023
1024 int expr_cmp_no_level_params::operator()(expr const &e1, expr const &e2) const {
1025     if (is_lt_no_level_params(e1, e2))
1026         return -1;
1027     else if (is_lt_no_level_params(e2, e1))
1028         return 1;
1029     else
1030         return 0;
1031 }
1032
1033 extern "C" uint8 lean_expr_quick_lt(b_obj_arg a, b_obj_arg b) {
1034     return is_lt(expr(a, true), expr(b, true), true, nullptr);
1035 }
1036
1037 extern "C" uint8 lean_expr_lt(b_obj_arg a, b_obj_arg b) {
1038     return is_lt(expr(a, true), expr(b, true), false, nullptr);
1039 }
1040 }  // namespace lean
1041 // ::::::::::::::
1042 // formatter.cpp
1043 // ::::::::::::::
1044 /*
1045 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
1046 Released under Apache 2.0 license as described in the file LICENSE.
1047
1048 Author: Leonardo de Moura
1049 */
```

```
#include <utility>

#include "library/formatter.h"

namespace lean {
static std::function<void(std::ostream &, expr const &e)> *g_print = nullptr;

void set_print_fn(std::function<void(std::ostream &, expr const &)> const &fn) {
    delete g_print;
    g_print = new std::function<void(std::ostream &, expr const &)>(fn);
}

std::ostream &operator<<(std::ostream &out, expr const &e) {
    if (g_print) {
        (*g_print)(out, e);
    } else {
        throw exception(
            "print function is not available, Lean was not initialized "
            "correctly");
    }
    return out;
}

void print(lean::expr const &a) { std::cout << a << std::endl; }

void initialize_formatter() {}

void finalize_formatter() { delete g_print; }
}  // namespace lean
// ::::::::::::::
// init_module.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include "library/annotation.h"
#include "library/class.h"
#include "library/constants.h"
#include "library/formatter.h"
#include "library/num.h"
#include "library/print.h"
#include "library/profiling.h"
#include "library/protected.h"
#include "library/time_task.h"
#include "library/trace.h"
#include "library/util.h"

namespace lean {
void initialize_library_core_module() {
    initialize_formatter();
    initialize_constants();
    initialize_profiling();
    initialize_trace();
}

void finalize_library_core_module() {
    finalize_trace();
    finalize_profiling();
    finalize_constants();
    finalize_formatter();
}

void initialize_library_module() {
    initialize_print();
    initialize_num();
    initialize_annotation();
    initialize_class();
```

```
1120     initialize_library_util();
1121     initialize_time_task();
1122 }
1123
1124 void finalize_library_module() {
1125     finalize_time_task();
1126     finalize_library_util();
1127     finalize_class();
1128     finalize_annotation();
1129     finalize_num();
1130     finalize_print();
1131 }
1132 }  // namespace lean
1133 // ::::::::::::::
1134 // max_sharing.cpp
1135 // ::::::::::::::
1136 /*
1137 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
1138 Released under Apache 2.0 license as described in the file LICENSE.
1139
1140 Author: Leonardo de Moura
1141 */
1142 #include <lean/interrupt.h>
1143
1144 #include <functional>
1145 #include <tuple>
1146 #include <unordered_set>
1147
1148 #include "library/max_sharing.h"
1149 #include "util/buffer.h"
1150
1151 namespace lean {
1152 /**
1153    \brief Implementation of the functional object for creating expressions with
1154    maximally shared sub-expressions.
1155 */
1156 struct max_sharing_fn::imp {
1157     typedef typename std::unordered_set<expr, expr_hash, is_bi_equal_proc>
1158         expr_cache;
1159     typedef typename std::unordered_set<level, level_hash> level_cache;
1160     expr_cache m_expr_cache;
1161     level_cache m_lvl_cache;
1162
1163     level apply(level const &l) {
1164         auto r = m_lvl_cache.find(l);
1165         if (r != m_lvl_cache.end()) return *r;
1166         level res;
1167         switch (l.kind()) {
1168             case level_kind::Zero:
1169             case level_kind::Param:
1170             case level_kind::MVar:
1171                 res = l;
1172                 break;
1173             case level_kind::Succ:
1174                 res = update_succ(l, apply(succ_of(l)));
1175                 break;
1176             case level_kind::Max:
1177                 res = update_max(l, apply(max_lhs(l)), apply(max_rhs(l)));
1178                 break;
1179             case level_kind::IMax:
1180                 res = update_max(l, apply(imax_lhs(l)), apply(imax_rhs(l)));
1181                 break;
1182         }
1183         m_lvl_cache.insert(res);
1184         return res;
1185     }
1186
1187     expr apply(expr const &a) {
1188         check_system("max_sharing");
1189         auto r = m_expr_cache.find(a);
```

```
1190         if (r != m_expr_cache.end()) return *r;
1191         expr res;
1192         switch (a.kind()) {
1193             case expr_kind::BVar:
1194             case expr_kind::Lit:
1195             case expr_kind::MVar:
1196             case expr_kind::FVar:
1197                 res = a;
1198                 break;
1199             case expr_kind::Const:
1200                 res = update_constant(
1201                     a, map(const_levels(a),
1202                            [&](level const &l) { return apply(l); }));
1203                 break;
1204             case expr_kind::Sort:
1205                 res = update_sort(a, apply(sort_level(a)));
1206                 break;
1207             case expr_kind::MData: {
1208                 expr new_e = apply(mdata_expr(a));
1209                 res = update_mdata(a, new_e);
1210                 break;
1211             }
1212             case expr_kind::Proj: {
1213                 expr new_e = apply(proj_expr(a));
1214                 res = update_proj(a, new_e);
1215                 break;
1216             }
1217             case expr_kind::App: {
1218                 expr new_f = apply(app_fn(a));
1219                 expr new_a = apply(app_arg(a));
1220                 res = update_app(a, new_f, new_a);
1221                 break;
1222             }
1223             case expr_kind::Lambda:
1224             case expr_kind::Pi: {
1225                 expr new_d = apply(binding_domain(a));
1226                 expr new_b = apply(binding_body(a));
1227                 res = update_binding(a, new_d, new_b);
1228                 break;
1229             }
1230             case expr_kind::Let: {
1231                 expr new_t = apply(let_type(a));
1232                 expr new_v = apply(let_value(a));
1233                 expr new_b = apply(let_body(a));
1234                 res = update_let(a, new_t, new_v, new_b);
1235                 break;
1236             }
1237         }
1238         m_expr_cache.insert(res);
1239         return res;
1240     }
1241
1242     expr operator()(expr const &a) { return apply(a); }
1243
1244     bool already_processed(expr const &a) const {
1245         auto r = m_expr_cache.find(a);
1246         return r != m_expr_cache.end() && is_eqp(*r, a);
1247     }
1248 };
1249
1250 max_sharing_fn::max_sharing_fn() : m_ptr(new imp) {}
1251 max_sharing_fn::~max_sharing_fn() {}
1252 expr max_sharing_fn::operator()(expr const &a) { return (*m_ptr)(a); }
1253 void max_sharing_fn::clear() { m_ptr->m_expr_cache.clear(); }
1254 bool max_sharing_fn::already_processed(expr const &a) const {
1255     return m_ptr->already_processed(a);
1256 }
1257
1258 expr max_sharing(expr const &a) { return max_sharing_fn::imp()(a); }
1259 }  // namespace lean
```

```
// ::::::::::::::
// module.cpp
// ::::::::::::::
/*
Copyright (c) 2014-2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Authors: Leonardo de Moura, Gabriel Ebner, Sebastian Ullrich
*/
#include <lean/compact.h>
#include <lean/hash.h>
#include <lean/interrupt.h>
#include <lean/io.h>
#include <lean/sstream.h>
#include <lean/thread.h>
#include <sys/stat.h>

#include <algorithm>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>

#include "library/constants.h"
#include "library/module.h"
#include "library/time_task.h"
#include "library/util.h"
#include "util/buffer.h"
#include "util/file_lock.h"
#include "util/io.h"
#include "util/name_map.h"

#if defined(__has_feature)
#if __has_feature(address_sanitizer)
#include <sanitizer/lsan_interface.h>
#endif
#endif

namespace lean {
// manually padded to multiple of word size, see `initialize_module`
static char const *g_olean_header = "oleanfile!!!!!!!";

extern "C" object *lean_save_module_data(object *fname, object *mdata,
                                         object *) {
    std::string olean_fn(string_cstr(fname));
    object_ref mdata_ref(mdata);
    try {
        exclusive_file_lock output_lock(olean_fn);
        std::ofstream out(olean_fn, std::ios_base::binary);
        if (out.fail()) {
            return io_result_mk_error(
                (sstream() << "failed to create file '" << olean_fn << "'")
                    .str());
        }
        object_compactor compactor;
        compactor(mdata_ref.raw());
        out.write(g_olean_header, strlen(g_olean_header));
        out.write(static_cast<char const *>(compactor.data()),
                  compactor.size());
        out.close();
        return io_result_mk_ok(box(0));
    } catch (exception &ex) {
        return io_result_mk_error(
            (sstream() << "failed to write '" << olean_fn << "': " << ex.what())
                .str());
    }
}

```

```
1330 extern "C" object *lean_read_module_data(object *fname, object *) {
1331     std::string olean_fn(string_cstr(fname));
1332     try {
1333         shared_file_lock olean_lock(olean_fn);
1334         std::ifstream in(olean_fn, std::ios_base::binary);
1335         if (in.fail()) {
1336             return io_result_mk_error(
1337                 (sstream() << "failed to open file '" << olean_fn << "'")
1338                     .str());
1339         }
1340         /* Get file size */
1341         in.seekg(0, in.end);
1342         size_t size = in.tellg();
1343         in.seekg(0);
1344         size_t header_size = strlen(g_olean_header);
1345         if (size < header_size) {
1346             return io_result_mk_error((sstream()
1347                                        << "failed to read file '" << olean_fn
1348                                        << "', invalid header")
1349                                           .str());
1350         }
1351         char *header = new char[header_size];
1352         in.read(header, header_size);
1353         if (strncmp(header, g_olean_header, header_size) != 0) {
1354             return io_result_mk_error((sstream()
1355                                        << "failed to read file '" << olean_fn
1356                                        << "', invalid header")
1357                                           .str());
1358         }
1359         delete[] header;
1360         // use `malloc` here as expected by `compacted_region`
1361         char *buffer = static_cast<char *>(malloc(size - header_size));
1362         in.read(buffer, size - header_size);
1363         if (!in) {
1364             return io_result_mk_error(
1365                 (sstream() << "failed to read file '" << olean_fn << "'")
1366                     .str());
1367         }
1368         in.close();
1369         compacted_region *region =
1370             new compacted_region(size - header_size, buffer);
1371 #if defined(__has_feature)
1372 #if __has_feature(address_sanitizer)
1373         // do not report as leak
1374         __lsan_ignore_object(region);
1375 #endif
1376 #endif
1377         object *mod = region->read();
1378         object *mod_region = alloc_cnstr(0, 2, 0);
1379         cnstr_set(mod_region, 0, mod);
1380         cnstr_set(mod_region, 1, box_size_t(reinterpret_cast<size_t>(region)));
1381         return io_result_mk_ok(mod_region);
1382     } catch (exception &ex) {
1383         return io_result_mk_error(
1384             (sstream() << "failed to read '" << olean_fn << "': " << ex.what())
1385                 .str());
1386     }
1387 }
1388
1389 /*
1390 @[export lean.write_module_core]
1391 def writeModule (env : Environment) (fname : String) : IO Unit := */
1392 extern "C" object *lean_write_module(object *env, object *fname, object *);
1393
1394 void write_module(environment const &env, std::string const &olean_fn) {
1395     consume_io_result(lean_write_module(env.to_obj_arg(), mk_string(olean_fn),
1396                                         io_mk_world()));
1397 }
1398 }  // namespace lean
1399 // :::::::::::::
```

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

Author: Leonardo de Moura
*/
#include "library/constants.h"
#include "library/num.h"
#include "library/util.h"

namespace lean {
bool is_const_app(expr const &e, name const &n, unsigned nargs) {
    expr const &f = get_app_fn(e);
    return is_constant(f) && const_name(f) == n && get_app_num_args(e) == nargs;
}

bool is_zero(expr const &e) {
    return is_const_app(e, get_has_zero_zero_name(), 2) ||
           is_constant(e, get_nat_zero_name());
}

bool is_one(expr const &e) {
    return is_const_app(e, get_has_one_one_name(), 2) ||
           (is_const_app(e, get_nat_succ_name(), 1) && is_zero(app_arg(e)));
}

optional<expr> is_bit0(expr const &e) {
    if (!is_const_app(e, get_bit0_name(), 3)) return none_expr();
    return some_expr(app_arg(e));
}

optional<expr> is_bit1(expr const &e) {
    if (!is_const_app(e, get_bit1_name(), 4)) return none_expr();
    return some_expr(app_arg(e));
}

optional<expr> is_neg(expr const &e) {
    if (!is_const_app(e, get_has_neg_neg_name(), 3)) return none_expr();
    return some_expr(app_arg(e));
}

optional<expr> is_of_nat(expr const &e) {
    if (!is_const_app(e, get_has_of_nat_of_nat_name(), 3)) return none_expr();
    return some_expr(app_arg(e));
}

optional<expr> unfold_num_app(environment const &env, expr const &e) {
    if (is_zero(e) || is_one(e) || is_bit0(e) || is_bit1(e)) {
        return unfold_app(env, e);
    } else {
        return none_expr();
    }
}

bool is_numeral_const_name(name const &n) {
    return n == get_has_zero_zero_name() || n == get_has_one_one_name() ||
           n == get_bit0_name() || n == get_bit1_name();
}

static bool is_num(expr const &e, bool first) {
    buffer<expr> args;
    expr const &f = get_app_args(e, args);
    if (!is_constant(f)) return false;
    if (const_name(f) == get_has_one_one_name())
        return args.size() == 2;
    else if (const_name(f) == get_has_zero_zero_name())
        return first && args.size() == 2;
    else if (const_name(f) == get_nat_zero_name())
```

```
1470         return first && args.size() == 0;
1471     else if (const_name(f) == get_bit0_name())
1472         return args.size() == 3 && is_num(args[2], false);
1473     else if (const_name(f) == get_bit1_name())
1474         return args.size() == 4 && is_num(args[3], false);
1475     return false;
1476 }
1477
1478 bool is_num(expr const &e) { return is_num(e, true); }
1479
1480 bool is_signed_num(expr const &e) {
1481     if (is_num(e))
1482         return true;
1483     else if (auto r = is_neg(e))
1484         return is_num(*r);
1485     else
1486         return false;
1487 }
1488
1489 static optional<mpz> to_num(expr const &e, bool first) {
1490     if (is_zero(e)) {
1491         return first ? some(mpz(0)) : optional<mpz>();
1492     } else if (is_one(e)) {
1493         return some(mpz(1));
1494     } else if (auto a = is_of_nat(e)) {
1495         return to_num(*a, false);
1496     } else if (is_lit(e) && lit_value(e).kind() == literal_kind::Nat) {
1497         return some(lit_value(e).get_nat().to_mpz());
1498     } else if (auto a = is_bit0(e)) {
1499         if (auto r = to_num(*a, false)) return some(2 * (*r));
1500     } else if (auto a = is_bit1(e)) {
1501         if (auto r = to_num(*a, false)) return some(2 * (*r) + 1);
1502     } else if (first) {
1503         if (auto a = is_neg(e)) {
1504             if (auto r = to_num(*a, false)) return some(neg(*r));
1505         }
1506     }
1507     return optional<mpz>();
1508 }
1509
1510 optional<mpz> to_num(expr const &e) { return to_num(e, true); }
1511
1512 bool is_num_leaf_constant(name const &n) {
1513     return n == get_has_zero_zero_name() || n == get_has_one_one_name();
1514 }
1515
1516 expr to_nat_expr_core(mpz const &n) {
1517     lean_assert(n >= 0);
1518     if (n == 1)
1519         return mk_nat_one();
1520     else if (n % mpz(2) == 0)
1521         return mk_nat_bit0(to_nat_expr(n / 2));
1522     else
1523         return mk_nat_bit1(to_nat_expr(n / 2));
1524 }
1525
1526 expr to_nat_expr(mpz const &n) {
1527     if (n == 0)
1528         return mk_nat_zero();
1529     else
1530         return to_nat_expr_core(n);
1531 }
1532
1533 void initialize_num() {}
1534 void finalize_num() {}
1535 }  // namespace lean
1536 // ::::::::::::::
1537 // print.cpp
1538 // ::::::::::::::
1539 /*
```

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

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

#include "kernel/environment.h"
#include "kernel/find_fn.h"
#include "kernel/instantiate.h"
#include "library/annotation.h"
#include "library/formatter.h"
#include "library/print.h"
#include "library/util.h"
#include "util/escaped.h"

namespace lean {
bool is_used_name(expr const &t, name const &n) {
    bool found = false;
    for_each(t, [&](expr const &e, unsigned) {
        if (found) return false;  // already found
        if ((is_constant(e) &&
             const_name(e).get_root() == n)  // t has a constant starting with n
            || (is_fvar(e) && fvar_name(e) == n)) {
            found = true;
            return false;  // found it
        }
        return true;  // continue search
    });
    return found;
}

name pick_unused_name(expr const &t, name const &s) {
    name r = s;
    unsigned i = 1;
    while (is_used_name(t, r)) {
        r = name(s).append_after(i);
        i++;
    }
    return r;
}

bool is_numerical_name(name n) {
    while (!n.is_atomic()) n = n.get_prefix();
    return n.is_numeral();
}

static name *g_M = nullptr;
static name *g_x = nullptr;

void initialize_print() {
    g_M = new name("M");
    mark_persistent(g_M->raw());
    g_x = new name("x");
    mark_persistent(g_x->raw());
}

void finalize_print() {
    delete g_M;
    delete g_x;
}

static name cleanup_name(name const &n) {
    if (is_numerical_name(n))
        return *g_x;
    else
        return n;
}

```

```
pair<expr, expr> binding_body_fresh(expr const &b, bool /* preserve_type */) {
    lean_assert(is_binding(b));
    name n = cleanup_name(binding_name(b));
    n = pick_unused_name(binding_body(b), n);
    expr c = mk_fvar(n);  // HACK
    return mk_pair(instantiate(binding_body(b), c), c);
}

pair<expr, expr> let_body_fresh(expr const &b, bool /* preserve_type */) {
    lean_assert(is_let(b));
    name n = cleanup_name(let_name(b));
    n = pick_unused_name(let_body(b), n);
    expr c = mk_fvar(n);  // HACK
    return mk_pair(instantiate(let_body(b), c), c);
}

name fix_name(name const &a) {
    if (a.is_atomic()) {
        if (a.is_numeral())
            return *g_M;
        else
            return a;
    } else {
        name p = fix_name(a.get_prefix());
        if (p == a.get_prefix())
            return a;
        else if (a.is_numeral())
            return name(p, a.get_numeral());
        else
            return name(p, a.get_string());
    }
}

/**
   \brief Very basic printer for expressions.
   It is mainly used when debugging code.
*/
struct print_expr_fn {
    std::ostream &m_out;

    std::ostream &out() { return m_out; }

    static bool is_atomic(expr const &a) {
        if (::lean::is_atomic(a)) return true;
        if (is_proj(a)) return is_atomic(proj_expr(a));
        return false;
    }

    void print_child(expr const &a) {
        if (is_atomic(a)) {
            print(a);
        } else {
            out() << "(";
            print(a);
            out() << ")";
        }
    }

    void print_sort(expr const &a) {
        if (is_zero(sort_level(a))) {
            out() << "Prop";
        } else if (is_one(sort_level(a))) {
            out() << "Type";
        } else if (is_succ(sort_level(a))) {
            out() << "Type.{" << succ_of(sort_level(a)) << "}";
        } else {
            out() << "Sort.{" << sort_level(a) << "}";
        }
    }

```

```
1680     void print_app(expr const &e) {
1681         expr const &f = app_fn(e);
1682         if (is_app(f))
1683             print(f);
1684         else
1685             print_child(f);
1686         out() << " ";
1687         print_child(app_arg(e));
1688     }
1689
1690     static bool is_arrow(expr const &t) {
1691         return lean::is_arrow(t) && binding_info(t) == binder_info::Default;
1692     }
1693
1694     void print_arrow_body(expr const &a) {
1695         if (is_atomic(a) || is_arrow(a))
1696             return print(a);
1697         else
1698             return print_child(a);
1699     }
1700
1701     void print_binding(char const *bname, expr e, bool is_lambda) {
1702         expr_kind k = e.kind();
1703         out() << bname;
1704         while (e.kind() == k && !is_arrow(e)) {
1705             out() << " ";
1706             auto p = binding_body_fresh(e);
1707             expr const &n = p.second;
1708             binder_info bi = binding_info(e);
1709             if (is_implicit(bi))
1710                 out() << "{";
1711             else if (is_inst_implicit(bi))
1712                 out() << "[";
1713             else if (is_strict_implicit(bi))
1714                 out() << "{{";
1715             else
1716                 out() << "(";
1717             out() << n << " : ";
1718             print(binding_domain(e));
1719             if (is_implicit(bi))
1720                 out() << "}";
1721             else if (is_inst_implicit(bi))
1722                 out() << "]";
1723             else if (is_strict_implicit(bi))
1724                 out() << "}}";
1725             else
1726                 out() << ")";
1727             e = p.first;
1728         }
1729         if (is_lambda)
1730             out() << " => ";
1731         else
1732             out() << ", ";
1733         print(e);
1734     }
1735
1736     void print_let(expr const &e) {
1737         auto p = let_body_fresh(e);
1738         out() << "let " << p.second << " : ";
1739         print(let_type(e));
1740         out() << " := ";
1741         print(let_value(e));
1742         out() << "; ";
1743         print(p.first);
1744     }
1745
1746     void print_const(expr const &a) {
1747         levels const &ls = const_levels(a);
1748         out() << const_name(a);
1749         if (!is_nil(ls)) {
```

```
1750            out() << ".{";
1751            bool first = true;
1752            for (auto l : ls) {
1753                if (first)
1754                    first = false;
1755                else
1756                    out() << " ";
1757                if (is_max(l) || is_imax(l))
1758                    out() << "(" << l << ")";
1759                else
1760                    out() << l;
1761            }
1762            out() << "}";
1763        }
1764    }
1765
1766    void print_mdata(expr const &a) {
1767        out() << "[mdata ";
1768        auto k = mdata_data(a);
1769        while (!empty(k)) {
1770            out() << head(k).fst() << ":";
1771            auto const &v = head(k).snd();
1772            switch (v.kind()) {
1773                case data_value_kind::Bool:
1774                    out() << v.get_bool();
1775                    break;
1776                case data_value_kind::Name:
1777                    out() << v.get_name();
1778                    break;
1779                case data_value_kind::Nat:
1780                    out() << v.get_nat();
1781                    break;
1782                case data_value_kind::String:
1783                    out() << escaped(v.get_string().data());
1784                    break;
1785            }
1786            out() << " ";
1787            k = tail(k);
1788        }
1789        print(mdata_expr(a));
1790        out() << "]";
1791    }
1792
1793    void print(expr const &a) {
1794        switch (a.kind()) {
1795            case expr_kind::MVar:
1796                out() << "?" << fix_name(mvar_name(a));
1797                break;
1798            case expr_kind::FVar:
1799                out() << fvar_name(a);
1800                break;
1801            case expr_kind::MData:
1802                print_mdata(a);
1803                break;
1804            case expr_kind::Proj:
1805                print_child(proj_expr(a));
1806                out() << "." << proj_idx(a).to_mpz();
1807                break;
1808            case expr_kind::BVar:
1809                out() << "#" << bvar_idx(a);
1810                break;
1811            case expr_kind::Const:
1812                print_const(a);
1813                break;
1814            case expr_kind::App:
1815                print_app(a);
1816                break;
1817            case expr_kind::Let:
1818                print_let(a);
1819                break;
```

```
1820             case expr_kind::Lambda:
1821                 print_binding("fun", a, true);
1822                 break;
1823             case expr_kind::Pi:
1824                 if (!is_arrow(a)) {
1825                     print_binding("forall", a, false);
1826                 } else {
1827                     print_child(binding_domain(a));
1828                     out() << " -> ";
1829                     print_arrow_body(lower_loose_bvars(binding_body(a), 1));
1830                 }
1831                 break;
1832             case expr_kind::Sort:
1833                 print_sort(a);
1834                 break;
1835             case expr_kind::Lit:
1836                 switch (lit_value(a).kind()) {
1837                     case literal_kind::Nat:
1838                         out() << lit_value(a).get_nat().to_mpz();
1839                         break;
1840                     case literal_kind::String: {
1841                         std::string val(
1842                             lit_value(a).get_string().to_std_string());
1843                         out() << "\"" << escaped(val.c_str()) << "\"";
1844                         break;  // HACK Lean string as C string
1845                     }
1846                 }
1847                 break;
1848         }
1849     }
1850
1851     print_expr_fn(std::ostream &out) : m_out(out) {}
1852
1853     void operator()(expr const &e) { print(e); }
1854 };
1855
1856 void init_default_print_fn() {
1857     set_print_fn([](std::ostream &out, expr const &e) {
1858         print_expr_fn pr(out);
1859         pr(e);
1860     });
1861 }
1862
1863 extern "C" object *lean_expr_dbg_to_string(b_obj_arg e) {
1864     std::ostringstream out;
1865     out << expr(e, true);
1866     return mk_string(out.str());
1867 }
1868 }  // namespace lean
1869 // ::::::::::::::
1870 // profiling.cpp
1871 // ::::::::::::::
1872 /*
1873 Copyright (c) 2017 Microsoft Corporation. All rights reserved.
1874 Released under Apache 2.0 license as described in the file LICENSE.
1875
1876 Author: Gabriel Ebner
1877 */
1878 #include "library/profiling.h"
1879 #include "util/option_declarations.h"
1880
1881 #ifndef LEAN_DEFAULT_PROFILER
1882 #define LEAN_DEFAULT_PROFILER false
1883 #endif
1884
1885 #ifndef LEAN_DEFAULT_PROFILER_THRESHOLD
1886 #define LEAN_DEFAULT_PROFILER_THRESHOLD 0
1887 #endif
1888
1889 namespace lean {
```

```
1890
1891 static name *g_profiler = nullptr;
1892 static name *g_profiler_threshold = nullptr;
1893
1894 bool get_profiler(options const &opts) {
1895     return opts.get_bool(*g_profiler, LEAN_DEFAULT_PROFILER);
1896 }
1897
1898 second_duration get_profiling_threshold(options const &opts) {
1899     return second_duration(
1900         static_cast<double>(opts.get_unsigned(
1901             *g_profiler_threshold, LEAN_DEFAULT_PROFILER_THRESHOLD)) /
1902         1000.0);
1903 }
1904
1905 void initialize_profiling() {
1906     g_profiler = new name{"profiler"};
1907     mark_persistent(g_profiler->raw());
1908     g_profiler_threshold = new name{"profiler", "threshold"};
1909     mark_persistent(g_profiler_threshold->raw());
1910     register_bool_option(*g_profiler, LEAN_DEFAULT_PROFILER,
1911                          "(profiler) profile tactics and vm_eval command");
1912     register_unsigned_option(*g_profiler_threshold,
1913                              LEAN_DEFAULT_PROFILER_THRESHOLD,
1914                              "(profiler) threshold in milliseconds, profiling "
1915                              "times under threshold will not be reported");
1916 }
1917
1918 void finalize_profiling() {
1919     delete g_profiler;
1920     delete g_profiler_threshold;
1921 }
1922
1923 }  // namespace lean
1924 // ::::::::::::::
1925 // projection.cpp
1926 // ::::::::::::::
1927 /*
1928 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
1929 Released under Apache 2.0 license as described in the file LICENSE.
1930
1931 Author: Leonardo de Moura
1932 */
1933 #include <lean/sstream.h>
1934
1935 #include <string>
1936
1937 #include "kernel/inductive.h"
1938 #include "kernel/instantiate.h"
1939 #include "kernel/kernel_exception.h"
1940 #include "library/projection.h"
1941 #include "library/util.h"
1942
1943 namespace lean {
1944 extern "C" object *lean_mk_projection_info(object *ctor_name, object *nparams,
1945                                            object *i, uint8 from_class);
1946 extern "C" uint8 lean_projection_info_from_class(object *info);
1947
1948 projection_info::projection_info(name const &c, unsigned nparams, unsigned i,
1949                                  bool inst_implicit)
1950     : object_ref(lean_mk_projection_info(c.to_obj_arg(),
1951                                          nat(nparams).to_obj_arg(),
1952                                          nat(i).to_obj_arg(), inst_implicit)) {}
1953
1954 bool projection_info::is_inst_implicit() const {
1955     return lean_projection_info_from_class(to_obj_arg());
1956 }
1957
1958 extern "C" object *lean_add_projection_info(object *env, object *p,
1959                                             object *ctor, object *nparams,
```

```
                                            object *i, uint8 fromClass);
extern "C" object *lean_get_projection_info(object *env, object *p);

environment save_projection_info(environment const &env, name const &p,
                                 name const &mk, unsigned nparams, unsigned i,
                                 bool inst_implicit) {
    return environment(lean_add_projection_info(
        env.to_obj_arg(), p.to_obj_arg(), mk.to_obj_arg(), mk_nat_obj(nparams),
        mk_nat_obj(i), inst_implicit));
}

optional<projection_info> get_projection_info(environment const &env,
                                              name const &p) {
    return to_optional<projection_info>(
        lean_get_projection_info(env.to_obj_arg(), p.to_obj_arg()));
}

/** \brief Return true iff the type named \c S can be viewed as
    a structure in the given environment.

    If not, generate an error message using \c pos. */
bool is_structure_like(environment const &env, name const &S) {
    constant_info S_info = env.get(S);
    if (!S_info.is_inductive()) return false;
    inductive_val S_val = S_info.to_inductive_val();
    return length(S_val.get_cnstrs()) == 1 && S_val.get_nindices() == 0;
}
}  // namespace lean
// ::::::::::::::
// protected.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include "library/protected.h"
#include "util/name_set.h"

namespace lean {
extern "C" object *lean_add_protected(object *env, object *n);
extern "C" uint8 lean_is_protected(object *env, object *n);

environment add_protected(environment const &env, name const &n) {
    return environment(lean_add_protected(env.to_obj_arg(), n.to_obj_arg()));
}

bool is_protected(environment const &env, name const &n) {
    return lean_is_protected(env.to_obj_arg(), n.to_obj_arg());
}

name get_protected_shortest_name(name const &n) {
    if (n.is_atomic() || n.get_prefix().is_atomic()) {
        return n;
    } else {
        name new_prefix =
            n.get_prefix().replace_prefix(n.get_prefix().get_prefix(), name());
        return n.replace_prefix(n.get_prefix(), new_prefix);
    }
}
}  // namespace lean
// ::::::::::::::
// reducible.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
```

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

Author: Leonardo de Moura
*/
#include <string>

#include "kernel/environment.h"
#include "library/reducible.h"

namespace lean {
extern "C" uint8 lean_get_reducibility_status(object *env, object *n);
extern "C" object *lean_set_reducibility_status(object *env, object *n,
                                                uint8 s);

environment set_reducible(environment const &env, name const &n,
                          reducible_status s, bool persistent) {
    if (!persistent)
        throw exception(
            "reducibility attributes must be persistent for now, we will relax "
            "this restriction in a near future");
    return environment(lean_set_reducibility_status(
        env.to_obj_arg(), n.to_obj_arg(), static_cast<uint8>(s)));
}

reducible_status get_reducible_status(environment const &env, name const &n) {
    return static_cast<reducible_status>(
        lean_get_reducibility_status(env.to_obj_arg(), n.to_obj_arg()));
}
}  // namespace lean
// ::::::::::::::
// replace_visitor.cpp
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include <tuple>

#include "kernel/abstract.h"
#include "kernel/instantiate.h"
#include "library/replace_visitor.h"

namespace lean {
expr replace_visitor::visit_sort(expr const &e) {
    lean_assert(is_sort(e));
    return e;
}
expr replace_visitor::visit_var(expr const &e) {
    lean_assert(is_var(e));
    return e;
}
expr replace_visitor::visit_lit(expr const &e) {
    lean_assert(is_lit(e));
    return e;
}
expr replace_visitor::visit_constant(expr const &e) {
    lean_assert(is_constant(e));
    return e;
}
expr replace_visitor::visit_meta(expr const &e) {
    lean_assert(is_mvar(e));
    return e;
}
expr replace_visitor::visit_fvar(expr const &e) {
    lean_assert(is_fvar(e));
    return e;
```

```
2100 }
2101 expr replace_visitor::visit_mdata(expr const &e) {
2102     return update_mdata(e, visit(mdata_expr(e)));
2103 }
2104 expr replace_visitor::visit_proj(expr const &e) {
2105     return update_proj(e, visit(proj_expr(e)));
2106 }
2107 expr replace_visitor::visit_app(expr const &e) {
2108     lean_assert(is_app(e));
2109     expr new_fn = visit(app_fn(e));
2110     expr new_arg = visit(app_arg(e));
2111     return update_app(e, new_fn, new_arg);
2112 }
2113 expr replace_visitor::visit_binding(expr const &e) {
2114     lean_assert(is_binding(e));
2115     expr new_d = visit(binding_domain(e));
2116     expr new_b = visit(binding_body(e));
2117     return update_binding(e, new_d, new_b);
2118 }
2119 expr replace_visitor::visit_lambda(expr const &e) { return visit_binding(e); }
2120 expr replace_visitor::visit_pi(expr const &e) { return visit_binding(e); }
2121 expr replace_visitor::visit_let(expr const &e) {
2122     lean_assert(is_let(e));
2123     expr new_t = visit(let_type(e));
2124     expr new_v = visit(let_value(e));
2125     expr new_b = visit(let_body(e));
2126     return update_let(e, new_t, new_v, new_b);
2127 }
2128 expr replace_visitor::save_result(expr const &e, expr &&r, bool shared) {
2129     if (shared) m_cache.insert(std::make_pair(e, r));
2130     return expr(r);
2131 }
2132 expr replace_visitor::visit(expr const &e) {
2133     check_system("expression replacer");
2134     bool shared = false;
2135     if (is_shared(e)) {
2136         shared = true;
2137         auto it = m_cache.find(e);
2138         if (it != m_cache.end()) return it->second;
2139     }
2140
2141     switch (e.kind()) {
2142         case expr_kind::Lit:
2143             return save_result(e, visit_lit(e), shared);
2144         case expr_kind::MData:
2145             return save_result(e, visit_mdata(e), shared);
2146         case expr_kind::Proj:
2147             return save_result(e, visit_proj(e), shared);
2148         case expr_kind::Sort:
2149             return save_result(e, visit_sort(e), shared);
2150         case expr_kind::Const:
2151             return save_result(e, visit_constant(e), shared);
2152         case expr_kind::BVar:
2153             return save_result(e, visit_var(e), shared);
2154         case expr_kind::MVar:
2155             return save_result(e, visit_meta(e), shared);
2156         case expr_kind::FVar:
2157             return save_result(e, visit_fvar(e), shared);
2158         case expr_kind::App:
2159             return save_result(e, visit_app(e), shared);
2160         case expr_kind::Lambda:
2161             return save_result(e, visit_lambda(e), shared);
2162         case expr_kind::Pi:
2163             return save_result(e, visit_pi(e), shared);
2164         case expr_kind::Let:
2165             return save_result(e, visit_let(e), shared);
2166     }
2167     lean_unreachable();  // LCOV_EXCL_LINE
2168 }
2169 }  // namespace lean
```

```
2170 // ::::::::::::::
2171 // sorry.cpp
2172 // ::::::::::::::
2173 /*
2174 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
2175 Released under Apache 2.0 license as described in the file LICENSE.
2176
2177 Author: Leonardo de Moura
2178 */
2179 #include <string>
2180
2181 #include "kernel/environment.h"
2182 #include "kernel/find_fn.h"
2183 #include "kernel/for_each_fn.h"
2184 #include "library/constants.h"
2185 #include "library/sorry.h"
2186 #include "library/util.h"
2187
2188 namespace lean {
2189 bool is_sorry(expr const &e) {
2190     return is_app_of(e, get_sorry_ax_name()) && get_app_num_args(e) >= 2;
2191 }
2192
2193 bool is_synthetic_sorry(expr const &e) {
2194     if (!is_sorry(e)) return false;
2195     buffer<expr> args;
2196     get_app_args(e, args);
2197     return is_constant(args[1], get_bool_true_name());
2198 }
2199
2200 bool has_synthetic_sorry(expr const &ex) {
2201     return static_cast<bool>(find(
2202         ex, [](expr const &e, unsigned) { return is_synthetic_sorry(e); }));
2203 }
2204
2205 bool has_sorry(expr const &ex) {
2206     return static_cast<bool>(
2207         find(ex, [](expr const &e, unsigned) { return is_sorry(e); }));
2208 }
2209
2210 bool has_sorry(declaration const &decl) {
2211     switch (decl.kind()) {
2212         case declaration_kind::Axiom:
2213             return has_sorry(decl.to_axiom_val().get_type());
2214         case declaration_kind::Definition:
2215             return has_sorry(decl.to_definition_val().get_type()) ||
2216                    has_sorry(decl.to_definition_val().get_value());
2217         case declaration_kind::Theorem:
2218             return has_sorry(decl.to_theorem_val().get_type()) ||
2219                    has_sorry(decl.to_theorem_val().get_value());
2220         case declaration_kind::Opaque:
2221             return has_sorry(decl.to_opaque_val().get_type()) ||
2222                    has_sorry(decl.to_opaque_val().get_value());
2223         case declaration_kind::Quot:
2224             return false;
2225         case declaration_kind::Inductive:
2226             return false;  // TODO(Leo):
2227         case declaration_kind::MutualDefinition:
2228             return false;  // TODO(Leo):
2229     }
2230     lean_unreachable();
2231 }
2232
2233 bool has_sorry(constant_info const &info) {
2234     return has_sorry(info.get_type()) ||
2235            (info.has_value() && has_sorry(info.get_value()));
2236 }
2237
2238 expr const &sorry_type(expr const &sry) {
2239     lean_assert(is_sorry(sry));
```

```
2240     buffer<expr> args;
2241     get_app_args(sry, args);
2242     return args[0];
2243 }
2244
2245 bool has_sorry(environment const &env) {
2246     bool found_sorry = false;
2247     env.for_each_constant([&](constant_info const &info) {
2248         if (!found_sorry && has_sorry(info)) found_sorry = true;
2249     });
2250     return found_sorry;
2251 }
2252 }  // namespace lean
2253 // ::::::::::::::
2254 // time_task.cpp
2255 // ::::::::::::::
2256 /*
2257 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
2258 Released under Apache 2.0 license as described in the file LICENSE.
2259
2260 Author: Sebastian Ullrich
2261 */
2262 #include <map>
2263 #include <string>
2264
2265 #include "library/time_task.h"
2266 #include "library/trace.h"
2267
2268 namespace lean {
2269
2270 static std::map<std::string, second_duration> *g_cum_times;
2271 static mutex *g_cum_times_mutex;
2272 LEAN_THREAD_PTR(time_task, g_current_time_task);
2273
2274 void report_profiling_time(std::string const &category, second_duration time) {
2275     lock_guard<mutex> _(*g_cum_times_mutex);
2276     (*g_cum_times)[category] += time;
2277 }
2278
2279 void display_cumulative_profiling_times(std::ostream &out) {
2280     if (g_cum_times->empty()) return;
2281     out << "cumulative profiling times:\n";
2282     for (auto const &p : *g_cum_times)
2283         out << "\t" << p.first << " " << display_profiling_time{p.second}
2284             << "\n";
2285 }
2286
2287 void initialize_time_task() {
2288     g_cum_times_mutex = new mutex;
2289     g_cum_times = new std::map<std::string, second_duration>;
2290 }
2291
2292 void finalize_time_task() {
2293     delete g_cum_times;
2294     delete g_cum_times_mutex;
2295 }
2296
2297 time_task::time_task(std::string const &category, options const &opts,
2298                      name decl)
2299     : m_category(category) {
2300     if (get_profiler(opts)) {
2301         m_timeit = optional<xtimeit>(
2302             get_profiling_threshold(opts),
2303             [=](second_duration duration) mutable {
2304                 tout() << m_category;
2305                 if (decl) tout() << " of " << decl;
2306                 tout() << " took " << display_profiling_time{duration} << "\n";
2307             });
2308         m_parent_task = g_current_time_task;
2309         g_current_time_task = this;
```

```
2310     }
2311 }
2312
2313 time_task::~time_task() {
2314     if (m_timeit) {
2315         g_current_time_task = m_parent_task;
2316         auto time = m_timeit->get_elapsed();
2317         report_profiling_time(m_category, time);
2318         if (m_parent_task)
2319             // do not report inclusive times
2320             report_profiling_time(m_parent_task->m_category, -time);
2321     }
2322 }
2323
2324 /* profileit {α : Type} (category : String) (opts : Options) (fn : Unit → α) : α
2325  */
2326 extern "C" obj_res lean_profileit(b_obj_arg category, b_obj_arg opts,
2327                                   obj_arg fn) {
2328     time_task t(string_to_std(category), TO_REF(options, opts));
2329     return apply_1(fn, box(0));
2330 }
2331 }  // namespace lean
2332 // ::::::::::::::
2333 // trace.cpp
2334 // ::::::::::::::
2335 /*
2336 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
2337 Released under Apache 2.0 license as described in the file LICENSE.
2338
2339 Author: Leonardo de Moura
2340 */
2341 #include <string>
2342 #include <vector>
2343
2344 #include "kernel/environment.h"
2345 #include "kernel/local_ctx.h"
2346 #include "library/trace.h"
2347 #include "util/io.h"
2348 #include "util/option_declarations.h"
2349
2350 namespace lean {
2351 static name_set *g_trace_classes = nullptr;
2352 static name_map<name_set> *g_trace_aliases = nullptr;
2353 MK_THREAD_LOCAL_GET_DEF(std::vector<name>, get_enabled_trace_classes);
2354 MK_THREAD_LOCAL_GET_DEF(std::vector<name>, get_disabled_trace_classes);
2355 LEAN_THREAD_PTR(environment, g_env);
2356 LEAN_THREAD_PTR(options, g_opts);
2357
2358 void register_trace_class(name const &n) {
2359     register_option(
2360         name("trace") + n, data_value_kind::Bool, "false",
2361         "(trace) enable/disable tracing for the given module and submodules");
2362     g_trace_classes->insert(n);
2363 }
2364
2365 void register_trace_class_alias(name const &n, name const &alias) {
2366     name_set new_s;
2367     if (auto s = g_trace_aliases->find(n)) new_s = *s;
2368     new_s.insert(alias);
2369     g_trace_aliases->insert(n, new_s);
2370 }
2371
2372 bool is_trace_enabled() { return !get_enabled_trace_classes().empty(); }
2373
2374 static void update_class(std::vector<name> &cs, name const &c) {
2375     if (std::find(cs.begin(), cs.end(), c) == cs.end()) {
2376         cs.push_back(c);
2377     }
2378 }
2379
```

```
2380 static void enable_trace_class(name const &c) {
2381     update_class(get_enabled_trace_classes(), c);
2382 }
2383
2384 static void disable_trace_class(name const &c) {
2385     update_class(get_disabled_trace_classes(), c);
2386 }
2387
2388 static bool is_trace_class_set_core(std::vector<name> const &cs,
2389                                     name const &n) {
2390     for (name const &p : cs) {
2391         if (is_prefix_of(p, n)) {
2392             return true;
2393         }
2394     }
2395     return false;
2396 }
2397
2398 static bool is_trace_class_set(std::vector<name> const &cs, name const &n) {
2399     if (is_trace_class_set_core(cs, n)) return true;
2400     auto it = n;
2401     while (true) {
2402         if (auto s = g_trace_aliases->find(it)) {
2403             bool found = false;
2404             s->for_each([&](name const &alias) {
2405                 if (!found && is_trace_class_set_core(cs, alias)) found = true;
2406             });
2407             if (found) return true;
2408         }
2409         if (it.is_atomic()) return false;
2410         it = it.get_prefix();
2411     }
2412 }
2413
2414 bool is_trace_class_enabled(name const &n) {
2415     if (!is_trace_enabled()) return false;
2416     if (is_trace_class_set(get_disabled_trace_classes(), n))
2417         return false;  // it was explicitly disabled
2418     return is_trace_class_set(get_enabled_trace_classes(), n);
2419 }
2420
2421 void scope_trace_env::init(environment *env, options *opts) {
2422     m_enable_sz = get_enabled_trace_classes().size();
2423     m_disable_sz = get_disabled_trace_classes().size();
2424     m_old_env = g_env;
2425     m_old_opts = g_opts;
2426     g_env = env;
2427     name trace("trace");
2428     if (opts && g_opts != opts) {
2429         opts->for_each([&](name const &n) {
2430             if (is_prefix_of(trace, n)) {
2431                 name cls = n.replace_prefix(trace, name());
2432                 if (opts->get_bool(n, false))
2433                     enable_trace_class(cls);
2434                 else
2435                     disable_trace_class(cls);
2436             }
2437         });
2438     }
2439     g_opts = opts;
2440 }
2441
2442 scope_trace_env::scope_trace_env(environment const &env, options const &o) {
2443     init(const_cast<environment *>(&env), const_cast<options *>(&o));
2444 }
2445
2446 scope_trace_env::~scope_trace_env() {
2447     g_env = const_cast<environment *>(m_old_env);
2448     g_opts = const_cast<options *>(m_old_opts);
2449     get_enabled_trace_classes().resize(m_enable_sz);
```

```
2450     get_disabled_trace_classes().resize(m_disable_sz);
2451 }
2452
2453 std::ostream &tout() { return std::cerr; }
2454
2455 std::ostream &operator<<(std::ostream &ios, tclass const &c) {
2456     ios << "[" << c.m_cls << "] ";
2457     return ios;
2458 }
2459
2460 void initialize_trace() {
2461     g_trace_classes = new name_set();
2462     g_trace_aliases = new name_map<name_set>();
2463
2464     register_trace_class(name{"debug"});
2465 }
2466
2467 void finalize_trace() {
2468     delete g_trace_classes;
2469     delete g_trace_aliases;
2470 }
2471
2472 /*
2473 @[export lean_mk_metavar_ctx]
2474 def mkMetavarContext : Unit → MetavarContext := fun _ => {}
2475 */
2476 extern "C" lean_object *lean_mk_metavar_ctx(lean_object *);
2477
2478 /*
2479 @[export lean_pp_expr]
2480 def ppExprLegacy (env : Environment) (mctx : MetavarContext) (lctx :
2481 LocalContext) (opts : Options) (e : Expr) : IO Format :=
2482 */
2483 extern "C" object *lean_pp_expr(object *env, object *mctx, object *lctx,
2484                                 object *opts, object *e, object *w);
2485
2486 /*
2487 @[export lean_format_pretty]
2488 def pretty (f : Format) (w : Nat := defWidth) : String :=
2489 */
2490 extern "C" object *lean_format_pretty(object *f, object *w);
2491
2492 std::string pp_expr(environment const &env, options const &opts,
2493                     expr const &e) {
2494     local_ctx lctx;
2495     object_ref fmt = get_io_result<object_ref>(lean_pp_expr(
2496         env.to_obj_arg(), lean_mk_metavar_ctx(lean_box(0)), lctx.to_obj_arg(),
2497         opts.to_obj_arg(), e.to_obj_arg(), io_mk_world()));
2498     string_ref str(
2499         lean_format_pretty(fmt.to_obj_arg(), lean_unsigned_to_nat(80)));
2500     return str.to_std_string();
2501 }
2502
2503 void trace_expr(environment const &env, options const &opts, expr const &e) {
2504     tout() << pp_expr(env, opts, e);
2505 }
2506
2507 std::string trace_pp_expr(expr const &e) { return pp_expr(*g_env, *g_opts, e); }
2508 }  // namespace lean
2509 // ::::::::::::::
2510 // util.cpp
2511 // ::::::::::::::
2512 /*
2513 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
2514 Released under Apache 2.0 license as described in the file LICENSE.
2515
2516 Author: Leonardo de Moura
2517 */
2518 #include <lean/version.h>
2519
```

```
#include <algorithm>
#include <string>

#include "githash.h"  // NOLINT
#include "kernel/abstract.h"
#include "kernel/find_fn.h"
#include "kernel/inductive.h"
#include "kernel/instantiate.h"
#include "kernel/type_checker.h"
#include "library/annotation.h"
#include "library/constants.h"
#include "library/num.h"
#include "library/projection.h"
#include "library/replace_visitor.h"
#include "library/suffixes.h"
#include "library/util.h"
#include "util/option_ref.h"

namespace lean {
name mk_unused_name(environment const &env, name const &n, unsigned &idx) {
    name curr = n;
    while (true) {
        if (!env.find(curr)) return curr;
        curr = n.append_after(idx);
        idx++;
    }
}

name mk_unused_name(environment const &env, name const &n) {
    unsigned idx = 1;
    return mk_unused_name(env, n, idx);
}

/** \brief Return the "arity" of the given type. The arity is the number of
 * nested pi-expressions. */
unsigned get_arity(expr type) {
    unsigned r = 0;
    while (is_pi(type)) {
        type = binding_body(type);
        r++;
    }
    return r;
}

optional<expr> is_optional_param(expr const &e) {
    if (is_app_of(e, get_opt_param_name(), 2)) {
        return some_expr(app_arg(e));
    } else {
        return none_expr();
    }
}

optional<expr_pair> is_auto_param(expr const &e) {
    if (is_app_of(e, get_auto_param_name(), 2)) {
        return optional<expr_pair>(app_arg(app_fn(e)), app_arg(e));
    } else {
        return optional<expr_pair>();
    }
}

name mk_fresh_lp_name(names const &lp_names) {
    name l("l");
    int i = 1;
    while (std::find(lp_names.begin(), lp_names.end(), l) != lp_names.end()) {
        l = name("l").append_after(i);
        i++;
    }
    return l;
}

```

```
2590 bool occurs(expr const &n, expr const &m) {
2591     return static_cast<bool>(
2592         find(m, [&](expr const &e, unsigned) { return n == e; }));
2593 }
2594
2595 bool occurs(name const &n, expr const &m) {
2596     return static_cast<bool>(find(m, [&](expr const &e, unsigned) {
2597         return is_constant(e) && const_name(e) == n;
2598     }));
2599 }
2600
2601 bool is_app_of(expr const &t, name const &f_name) {
2602     expr const &fn = get_app_fn(t);
2603     return is_constant(fn) && const_name(fn) == f_name;
2604 }
2605
2606 bool is_app_of(expr const &t, name const &f_name, unsigned nargs) {
2607     expr const &fn = get_app_fn(t);
2608     return is_constant(fn) && const_name(fn) == f_name &&
2609            get_app_num_args(t) == nargs;
2610 }
2611
2612 expr consume_auto_opt_param(expr const &type) {
2613     if (is_app_of(type, get_auto_param_name(), 2) ||
2614         is_app_of(type, get_opt_param_name(), 2)) {
2615         return app_arg(app_fn(type));
2616     } else {
2617         return type;
2618     }
2619 }
2620
2621 optional<expr> unfold_term(environment const &env, expr const &e) {
2622     expr const &f = get_app_fn(e);
2623     if (!is_constant(f)) return none_expr();
2624     auto decl = env.find(const_name(f));
2625     if (!decl || !decl->has_value()) return none_expr();
2626     expr d = instantiate_value_lparams(*decl, const_levels(f));
2627     buffer<expr> args;
2628     get_app_rev_args(e, args);
2629     return some_expr(apply_beta(d, args.size(), args.data()));
2630 }
2631
2632 optional<expr> unfold_app(environment const &env, expr const &e) {
2633     if (!is_app(e)) return none_expr();
2634     return unfold_term(env, e);
2635 }
2636
2637 optional<level> dec_level(level const &l) {
2638     switch (kind(l)) {
2639         case level_kind::Zero:
2640         case level_kind::Param:
2641         case level_kind::MVar:
2642             return none_level();
2643         case level_kind::Succ:
2644             return some_level(succ_of(l));
2645         case level_kind::Max:
2646             if (auto lhs = dec_level(max_lhs(l))) {
2647                 if (auto rhs = dec_level(max_rhs(l))) {
2648                     return some_level(mk_max(*lhs, *rhs));
2649                 }
2650             }
2651             return none_level();
2652         case level_kind::IMax:
2653             // Remark: the following mk_max is not a typo. The following
2654             // assertion justifies it.
2655             if (auto lhs = dec_level(imax_lhs(l))) {
2656                 if (auto rhs = dec_level(imax_rhs(l))) {
2657                     return some_level(mk_max(*lhs, *rhs));
2658                 }
2659             }
```

```
2660             return none_level();
2661     }
2662     lean_unreachable();  // LCOV_EXCL_LINE
2663 }
2664
2665 /** \brief Return true if environment has a constructor named \c c that returns
2666     an element of the inductive datatype named \c I, and \c c must have \c
2667    nparams parameters. */
2668 bool has_constructor(environment const &env, name const &c, name const &I,
2669                      unsigned nparams) {
2670     auto d = env.find(c);
2671     if (!d || d->has_value()) return false;
2672     expr type = d->get_type();
2673     unsigned i = 0;
2674     while (is_pi(type)) {
2675         i++;
2676         type = binding_body(type);
2677     }
2678     if (i != nparams) return false;
2679     type = get_app_fn(type);
2680     return is_constant(type) && const_name(type) == I;
2681 }
2682
2683 bool has_punit_decls(environment const &env) {
2684     return has_constructor(env, get_punit_unit_name(), get_punit_name(), 0);
2685 }
2686
2687 bool has_eq_decls(environment const &env) {
2688     return has_constructor(env, get_eq_refl_name(), get_eq_name(), 2);
2689 }
2690
2691 bool has_heq_decls(environment const &env) {
2692     return has_constructor(env, get_heq_refl_name(), get_heq_name(), 2);
2693 }
2694
2695 bool has_pprod_decls(environment const &env) {
2696     return has_constructor(env, get_pprod_mk_name(), get_pprod_name(), 4);
2697 }
2698
2699 bool has_and_decls(environment const &env) {
2700     return has_constructor(env, get_and_intro_name(), get_and_name(), 4);
2701 }
2702
2703 /* n is considered to be recursive if it is an inductive datatype and
2704    1) It has a constructor that takes n as an argument
2705    2) It is part of a mutually recursive declaration, and some constructor
2706       of an inductive datatype takes another inductive datatype from the
2707       same declaration as an argument. */
2708 bool is_recursive_datatype(environment const &env, name const &n) {
2709     constant_info info = env.get(n);
2710     return info.is_inductive() && info.to_inductive_val().is_rec();
2711 }
2712
2713 level get_datatype_level(expr const &ind_type) {
2714     expr it = ind_type;
2715     while (is_pi(it)) it = binding_body(it);
2716     if (is_sort(it)) {
2717         return sort_level(it);
2718     } else {
2719         throw exception("invalid inductive datatype type");
2720     }
2721 }
2722
2723 expr update_result_sort(expr t, level const &l) {
2724     if (is_pi(t)) {
2725         return update_binding(t, binding_domain(t),
2726                               update_result_sort(binding_body(t), l));
2727     } else if (is_sort(t)) {
2728         return update_sort(t, l);
2729     } else {
```

```
2730         lean_unreachable();
2731     }
2732 }
2733
2734 bool is_inductive_predicate(environment const &env, name const &n) {
2735     constant_info info = env.get(n);
2736     if (!info.is_inductive()) return false;
2737     return is_zero(get_datatype_level(env.get(n).get_type()));
2738 }
2739
2740 bool can_elim_to_type(environment const &env, name const &n) {
2741     constant_info ind_info = env.get(n);
2742     if (!ind_info.is_inductive()) return false;
2743     constant_info rec_info = env.get(mk_rec_name(n));
2744     return rec_info.get_num_lparams() > ind_info.get_num_lparams();
2745 }
2746
2747 void get_constructor_names(environment const &env, name const &n,
2748                            buffer<name> &result) {
2749     constant_info info = env.get(n);
2750     if (!info.is_inductive()) return;
2751     to_buffer(info.to_inductive_val().get_cnstrs(), result);
2752 }
2753
2754 optional<name> is_constructor_app(environment const &env, expr const &e) {
2755     expr const &fn = get_app_fn(e);
2756     if (is_constant(fn)) {
2757         if (is_constructor(env, const_name(fn)))
2758             return optional<name>(const_name(fn));
2759     }
2760     return optional<name>();
2761 }
2762
2763 optional<name> is_constructor_app_ext(environment const &env, expr const &e) {
2764     if (auto r = is_constructor_app(env, e)) return r;
2765     expr const &f = get_app_fn(e);
2766     if (!is_constant(f)) return optional<name>();
2767     optional<constant_info> info = env.find(const_name(f));
2768     if (!info || !info->has_value()) return optional<name>();
2769     expr val = info->get_value();
2770     expr const *it = &val;
2771     while (is_lambda(*it)) it = &binding_body(*it);
2772     return is_constructor_app_ext(env, *it);
2773 }
2774
2775 static name *g_util_fresh = nullptr;
2776
2777 void get_constructor_relevant_fields(environment const &env, name const &n,
2778                                      buffer<bool> &result) {
2779     constant_info info = env.get(n);
2780     lean_assert(info.is_constructor());
2781     constructor_val val = info.to_constructor_val();
2782     expr type = info.get_type();
2783     name I_name = val.get_induct();
2784     unsigned nparams = val.get_nparams();
2785     local_ctx lctx;
2786     name_generator ngen(*g_util_fresh);
2787     buffer<expr> telescope;
2788     to_telescope(env, lctx, ngen, type, telescope);
2789     lean_assert(telescope.size() >= nparams);
2790     for (unsigned i = nparams; i < telescope.size(); i++) {
2791         expr ftype = lctx.get_type(telescope[i]);
2792         if (type_checker(env, lctx).is_prop(ftype)) {
2793             result.push_back(false);
2794         } else {
2795             buffer<expr> tmp;
2796             expr n_ftype = to_telescope(env, lctx, ngen, ftype, tmp);
2797             result.push_back(!is_sort(n_ftype) &&
2798                              !type_checker(env, lctx).is_prop(n_ftype));
2799         }
```

```
2800     }
2801 }
2802
2803 unsigned get_num_constructors(environment const &env, name const &n) {
2804     constant_info info = env.get(n);
2805     lean_assert(info.is_inductive());
2806     return length(info.to_inductive_val().get_cnstrs());
2807 }
2808
2809 unsigned get_constructor_idx(environment const &env, name const &n) {
2810     constant_info info = env.get(n);
2811     lean_assert(info.is_constructor());
2812     constructor_val val = info.to_constructor_val();
2813     name I_name = val.get_induct();
2814     buffer<name> cnames;
2815     get_constructor_names(env, I_name, cnames);
2816     unsigned r = 0;
2817     for (name const &cname : cnames) {
2818         if (cname == n) return r;
2819         r++;
2820     }
2821     lean_unreachable();
2822 }
2823
2824 name get_constructor_inductive_type(environment const &env,
2825                                     name const &ctor_name) {
2826     constant_info info = env.get(ctor_name);
2827     lean_assert(info.is_constructor());
2828     constructor_val val = info.to_constructor_val();
2829     return val.get_induct();
2830 }
2831
2832 expr instantiate_lparam(expr const &e, name const &p, level const &l) {
2833     return instantiate_lparams(e, names(p), levels(l));
2834 }
2835
2836 expr to_telescope(bool pi, local_ctx &lctx, name_generator &ngen, expr e,
2837                   buffer<expr> &telescope, optional<binder_info> const &binfo) {
2838     while ((pi && is_pi(e)) || (!pi && is_lambda(e))) {
2839         expr local;
2840         if (binfo)
2841             local = lctx.mk_local_decl(ngen, binding_name(e), binding_domain(e),
2842                                        *binfo);
2843         else
2844             local = lctx.mk_local_decl(ngen, binding_name(e), binding_domain(e),
2845                                        binding_info(e));
2846         telescope.push_back(local);
2847         e = instantiate(binding_body(e), local);
2848     }
2849     return e;
2850 }
2851
2852 expr to_telescope(local_ctx &lctx, name_generator &ngen, expr const &type,
2853                   buffer<expr> &telescope, optional<binder_info> const &binfo) {
2854     return to_telescope(true, lctx, ngen, type, telescope, binfo);
2855 }
2856
2857 expr to_telescope(environment const &env, local_ctx &lctx, name_generator &ngen,
2858                   expr type, buffer<expr> &telescope,
2859                   optional<binder_info> const &binfo) {
2860     expr new_type = type_checker(env, lctx).whnf(type);
2861     while (is_pi(new_type)) {
2862         type = new_type;
2863         expr local;
2864         if (binfo)
2865             local = lctx.mk_local_decl(ngen, binding_name(type),
2866                                        binding_domain(type), *binfo);
2867         else
2868             local =
2869                 lctx.mk_local_decl(ngen, binding_name(type),
```

```
2870                                     binding_domain(type), binding_info(type));
2871         telescope.push_back(local);
2872         type = instantiate(binding_body(type), local);
2873         new_type = type_checker(env, lctx).whnf(type);
2874     }
2875     return type;
2876 }
2877
2878 /* ------------------------------------------
2879
2880    Helper functions for creating basic operations
2881
2882    ------------------------------------------ */
2883 static expr *g_true = nullptr;
2884 static expr *g_true_intro = nullptr;
2885 static expr *g_and = nullptr;
2886 static expr *g_and_intro = nullptr;
2887 static expr *g_and_left = nullptr;
2888 static expr *g_and_right = nullptr;
2889
2890 expr mk_true() { return *g_true; }
2891
2892 bool is_true(expr const &e) { return e == *g_true; }
2893
2894 expr mk_true_intro() { return *g_true_intro; }
2895
2896 bool is_and(expr const &e) { return is_app_of(e, get_and_name(), 2); }
2897
2898 bool is_and(expr const &e, expr &arg1, expr &arg2) {
2899     if (is_and(e)) {
2900         arg1 = app_arg(app_fn(e));
2901         arg2 = app_arg(e);
2902         return true;
2903     } else {
2904         return false;
2905     }
2906 }
2907
2908 expr mk_and(expr const &a, expr const &b) { return mk_app(*g_and, a, b); }
2909
2910 expr mk_unit(level const &l) { return mk_constant(get_punit_name(), {l}); }
2911
2912 expr mk_unit_mk(level const &l) {
2913     return mk_constant(get_punit_unit_name(), {l});
2914 }
2915
2916 static expr *g_unit = nullptr;
2917 static expr *g_unit_mk = nullptr;
2918
2919 expr mk_unit() { return *g_unit; }
2920
2921 expr mk_unit_mk() { return *g_unit_mk; }
2922
2923 static expr *g_nat = nullptr;
2924 static expr *g_nat_zero = nullptr;
2925 static expr *g_nat_one = nullptr;
2926 static expr *g_nat_bit0_fn = nullptr;
2927 static expr *g_nat_bit1_fn = nullptr;
2928 static expr *g_nat_add_fn = nullptr;
2929
2930 static void initialize_nat() {
2931     g_nat = new expr(mk_constant(get_nat_name()));
2932     mark_persistent(g_nat->raw());
2933     g_nat_zero = new expr(
2934         mk_app(mk_constant(get_has_zero_zero_name(), {mk_level_zero()}),
2935                {*g_nat, mk_constant(get_nat_has_zero_name())}));
2936     mark_persistent(g_nat_zero->raw());
2937     g_nat_one =
2938         new expr(mk_app(mk_constant(get_has_one_one_name(), {mk_level_zero()}),
2939                         {*g_nat, mk_constant(get_nat_has_one_name())}));
```

```
2940     mark_persistent(g_nat_one->raw());
2941     g_nat_bit0_fn =
2942         new expr(mk_app(mk_constant(get_bit0_name(), {mk_level_zero()}),
2943                         {*g_nat, mk_constant(get_nat_has_add_name())}));
2944     mark_persistent(g_nat_bit0_fn->raw());
2945     g_nat_bit1_fn =
2946         new expr(mk_app(mk_constant(get_bit1_name(), {mk_level_zero()}),
2947                         {*g_nat, mk_constant(get_nat_has_one_name()),
2948                          mk_constant(get_nat_has_add_name())}));
2949     mark_persistent(g_nat_bit1_fn->raw());
2950     g_nat_add_fn =
2951         new expr(mk_app(mk_constant(get_has_add_add_name(), {mk_level_zero()}),
2952                         {*g_nat, mk_constant(get_nat_has_add_name())}));
2953     mark_persistent(g_nat_add_fn->raw());
2954 }
2955
2956 static void finalize_nat() {
2957     delete g_nat;
2958     delete g_nat_zero;
2959     delete g_nat_one;
2960     delete g_nat_bit0_fn;
2961     delete g_nat_bit1_fn;
2962     delete g_nat_add_fn;
2963 }
2964
2965 expr mk_nat_type() { return *g_nat; }
2966 bool is_nat_type(expr const &e) { return e == *g_nat; }
2967 expr mk_nat_zero() { return *g_nat_zero; }
2968 expr mk_nat_one() { return *g_nat_one; }
2969 expr mk_nat_bit0(expr const &e) { return mk_app(*g_nat_bit0_fn, e); }
2970 expr mk_nat_bit1(expr const &e) { return mk_app(*g_nat_bit1_fn, e); }
2971 expr mk_nat_add(expr const &e1, expr const &e2) {
2972     return mk_app(*g_nat_add_fn, e1, e2);
2973 }
2974
2975 static expr *g_int = nullptr;
2976
2977 static void initialize_int() {
2978     g_int = new expr(mk_constant(get_int_name()));
2979     mark_persistent(g_int->raw());
2980 }
2981
2982 static void finalize_int() { delete g_int; }
2983
2984 expr mk_int_type() { return *g_int; }
2985 bool is_int_type(expr const &e) { return e == *g_int; }
2986
2987 static expr *g_char = nullptr;
2988
2989 expr mk_char_type() { return *g_char; }
2990
2991 static void initialize_char() {
2992     g_char = new expr(mk_constant(get_char_name()));
2993     mark_persistent(g_char->raw());
2994 }
2995
2996 static void finalize_char() { delete g_char; }
2997
2998 expr mk_unit(level const &l, bool prop) {
2999     return prop ? mk_true() : mk_unit(l);
3000 }
3001 expr mk_unit_mk(level const &l, bool prop) {
3002     return prop ? mk_true_intro() : mk_unit_mk(l);
3003 }
3004
3005 bool is_ite(expr const &e) { return is_app_of(e, get_ite_name(), 5); }
3006
3007 bool is_ite(expr const &e, expr &c, expr &H, expr &A, expr &t, expr &f) {
3008     if (is_ite(e)) {
3009         buffer<expr> args;
```

```
3010         get_app_args(e, args);
3011         lean_assert(args.size() == 5);
3012         c = args[0];
3013         H = args[1];
3014         A = args[2];
3015         t = args[3];
3016         f = args[4];
3017         return true;
3018     } else {
3019         return false;
3020     }
3021 }
3022
3023 bool is_iff(expr const &e) { return is_app_of(e, get_iff_name(), 2); }
3024
3025 bool is_iff(expr const &e, expr &lhs, expr &rhs) {
3026     if (!is_iff(e)) return false;
3027     lhs = app_arg(app_fn(e));
3028     rhs = app_arg(e);
3029     return true;
3030 }
3031 expr mk_iff(expr const &lhs, expr const &rhs) {
3032     return mk_app(mk_constant(get_iff_name()), lhs, rhs);
3033 }
3034 expr mk_iff_refl(expr const &a) {
3035     return mk_app(mk_constant(get_iff_refl_name()), a);
3036 }
3037 expr mk_propext(expr const &lhs, expr const &rhs, expr const &iff_pr) {
3038     return mk_app(mk_constant(get_propext_name()), lhs, rhs, iff_pr);
3039 }
3040
3041 bool is_eq_ndrec_core(expr const &e) {
3042     expr const &fn = get_app_fn(e);
3043     return is_constant(fn) && const_name(fn) == get_eq_ndrec_name();
3044 }
3045
3046 bool is_eq_ndrec(expr const &e) {
3047     expr const &fn = get_app_fn(e);
3048     if (!is_constant(fn)) return false;
3049     return const_name(fn) == get_eq_ndrec_name();
3050 }
3051
3052 bool is_eq_rec(expr const &e) {
3053     expr const &fn = get_app_fn(e);
3054     if (!is_constant(fn)) return false;
3055     return const_name(fn) == get_eq_rec_name();
3056 }
3057
3058 bool is_eq(expr const &e) { return is_app_of(e, get_eq_name(), 3); }
3059
3060 bool is_eq(expr const &e, expr &lhs, expr &rhs) {
3061     if (!is_eq(e)) return false;
3062     lhs = app_arg(app_fn(e));
3063     rhs = app_arg(e);
3064     return true;
3065 }
3066
3067 bool is_eq(expr const &e, expr &A, expr &lhs, expr &rhs) {
3068     if (!is_eq(e)) return false;
3069     A = app_arg(app_fn(app_fn(e)));
3070     lhs = app_arg(app_fn(e));
3071     rhs = app_arg(e);
3072     return true;
3073 }
3074
3075 bool is_eq_a_a(expr const &e) {
3076     if (!is_eq(e)) return false;
3077     expr lhs = app_arg(app_fn(e));
3078     expr rhs = app_arg(e);
3079     return lhs == rhs;
```

```
3080 }
3081
3082 bool is_heq(expr const &e) { return is_app_of(e, get_heq_name(), 4); }
3083
3084 bool is_heq(expr const &e, expr &A, expr &lhs, expr &B, expr &rhs) {
3085     if (is_heq(e)) {
3086         buffer<expr> args;
3087         get_app_args(e, args);
3088         lean_assert(args.size() == 4);
3089         A = args[0];
3090         lhs = args[1];
3091         B = args[2];
3092         rhs = args[3];
3093         return true;
3094     } else {
3095         return false;
3096     }
3097 }
3098
3099 bool is_heq(expr const &e, expr &lhs, expr &rhs) {
3100     expr A, B;
3101     return is_heq(e, A, lhs, B, rhs);
3102 }
3103
3104 expr mk_false() { return mk_constant(get_false_name()); }
3105
3106 expr mk_empty() { return mk_constant(get_empty_name()); }
3107
3108 bool is_false(expr const &e) {
3109     return is_constant(e) && const_name(e) == get_false_name();
3110 }
3111
3112 bool is_empty(expr const &e) {
3113     return is_constant(e) && const_name(e) == get_empty_name();
3114 }
3115
3116 bool is_or(expr const &e) { return is_app_of(e, get_or_name(), 2); }
3117
3118 bool is_or(expr const &e, expr &A, expr &B) {
3119     if (is_or(e)) {
3120         A = app_arg(app_fn(e));
3121         B = app_arg(e);
3122         return true;
3123     } else {
3124         return false;
3125     }
3126 }
3127
3128 bool is_not(expr const &e, expr &a) {
3129     if (is_app_of(e, get_not_name(), 1)) {
3130         a = app_arg(e);
3131         return true;
3132     } else if (is_pi(e) && is_false(binding_body(e))) {
3133         a = binding_domain(e);
3134         return true;
3135     } else {
3136         return false;
3137     }
3138 }
3139
3140 bool is_not_or_ne(expr const &e, expr &a) {
3141     if (is_not(e, a)) {
3142         return true;
3143     } else if (is_app_of(e, get_ne_name(), 3)) {
3144         buffer<expr> args;
3145         expr const &fn = get_app_args(e, args);
3146         expr new_fn = mk_constant(get_eq_name(), const_levels(fn));
3147         a = mk_app(new_fn, args);
3148         return true;
3149     } else {
```

```
3150         return false;
3151     }
3152 }
3153
3154 expr mk_not(expr const &e) { return mk_app(mk_constant(get_not_name()), e); }
3155
3156 bool is_exists(expr const &e, expr &A, expr &p) {
3157     if (is_app_of(e, get_exists_name(), 2)) {
3158         A = app_arg(app_fn(e));
3159         p = app_arg(e);
3160         return true;
3161     } else {
3162         return false;
3163     }
3164 }
3165
3166 bool is_exists(expr const &e) { return is_app_of(e, get_exists_name(), 2); }
3167
3168 optional<expr> get_binary_op(expr const &e) {
3169     if (!is_app(e) || !is_app(app_fn(e))) return none_expr();
3170     return some_expr(app_fn(app_fn(e)));
3171 }
3172
3173 optional<expr> get_binary_op(expr const &e, expr &arg1, expr &arg2) {
3174     if (auto op = get_binary_op(e)) {
3175         arg1 = app_arg(app_fn(e));
3176         arg2 = app_arg(e);
3177         return some_expr(*op);
3178     } else {
3179         return none_expr();
3180     }
3181 }
3182
3183 expr mk_nary_app(expr const &op, buffer<expr> const &nary_args) {
3184     return mk_nary_app(op, nary_args.size(), nary_args.data());
3185 }
3186
3187 expr mk_nary_app(expr const &op, unsigned num_nary_args,
3188                  expr const *nary_args) {
3189     lean_assert(num_nary_args >= 2);
3190     // f x1 x2 x3 ==> f x1 (f x2 x3)
3191     expr e =
3192         mk_app(op, nary_args[num_nary_args - 2], nary_args[num_nary_args - 1]);
3193     for (int i = num_nary_args - 3; i >= 0; --i) {
3194         e = mk_app(op, nary_args[i], e);
3195     }
3196     return e;
3197 }
3198
3199 bool is_annotated_lamba(expr const &e) {
3200     return is_lambda(e) ||
3201            (is_annotation(e) && is_lambda(get_nested_annotation_arg(e)));
3202 }
3203
3204 bool is_annotated_head_beta(expr const &t) {
3205     return is_app(t) && is_annotated_lamba(get_app_fn(t));
3206 }
3207
3208 expr annotated_head_beta_reduce(expr const &t) {
3209     if (!is_annotated_head_beta(t)) {
3210         return t;
3211     } else {
3212         buffer<expr> args;
3213         expr f = get_app_rev_args(t, args);
3214         if (is_annotation(f)) f = get_nested_annotation_arg(f);
3215         lean_assert(is_lambda(f));
3216         return annotated_head_beta_reduce(
3217             apply_beta(f, args.size(), args.data()));
3218     }
3219 }
```

```
3220
3221 expr try_eta(expr const &e) {
3222     if (is_lambda(e)) {
3223         expr const &b = binding_body(e);
3224         if (is_lambda(b)) {
3225             expr new_b = try_eta(b);
3226             if (is_eqp(b, new_b)) {
3227                 return e;
3228             } else if (is_app(new_b) && is_var(app_arg(new_b), 0) &&
3229                        !has_loose_bvar(app_fn(new_b), 0)) {
3230                 return lower_loose_bvars(app_fn(new_b), 1);
3231             } else {
3232                 return update_binding(e, binding_domain(e), new_b);
3233             }
3234         } else if (is_app(b) && is_var(app_arg(b), 0) &&
3235                    !has_loose_bvar(app_fn(b), 0)) {
3236             return lower_loose_bvars(app_fn(b), 1);
3237         } else {
3238             return e;
3239         }
3240     } else {
3241         return e;
3242     }
3243 }
3244
3245 template <bool Eta, bool Beta>
3246 class eta_beta_reduce_fn : public replace_visitor {
3247    public:
3248     virtual expr visit_app(expr const &e) override {
3249         expr e1 = replace_visitor::visit_app(e);
3250         if (Beta && is_head_beta(e1)) {
3251             return visit(head_beta_reduce(e1));
3252         } else {
3253             return e1;
3254         }
3255     }
3256
3257     virtual expr visit_lambda(expr const &e) override {
3258         expr e1 = replace_visitor::visit_lambda(e);
3259         if (Eta) {
3260             while (true) {
3261                 expr e2 = try_eta(e1);
3262                 if (is_eqp(e1, e2))
3263                     return e1;
3264                 else
3265                     e1 = e2;
3266             }
3267         } else {
3268             return e1;
3269         }
3270     }
3271 };
3272
3273 expr beta_reduce(expr t) { return eta_beta_reduce_fn<false, true>()(t); }
3274
3275 expr eta_reduce(expr t) { return eta_beta_reduce_fn<true, false>()(t); }
3276
3277 expr beta_eta_reduce(expr t) { return eta_beta_reduce_fn<true, true>()(t); }
3278
3279 expr infer_implicit_params(expr const &type, unsigned nparams,
3280                            implicit_infer_kind k) {
3281     switch (k) {
3282         case implicit_infer_kind::Implicit: {
3283             bool strict = true;
3284             return infer_implicit(type, nparams, strict);
3285         }
3286         case implicit_infer_kind::RelaxedImplicit: {
3287             bool strict = false;
3288             return infer_implicit(type, nparams, strict);
3289         }
```

```
3290     }
3291     lean_unreachable();  // LCOV_EXCL_LINE
3292 }
3293
3294 static expr *g_bool = nullptr;
3295 static expr *g_bool_true = nullptr;
3296 static expr *g_bool_false = nullptr;
3297
3298 void initialize_bool() {
3299     g_bool = new expr(mk_constant(get_bool_name()));
3300     mark_persistent(g_bool->raw());
3301     g_bool_false = new expr(mk_constant(get_bool_false_name()));
3302     mark_persistent(g_bool_false->raw());
3303     g_bool_true = new expr(mk_constant(get_bool_true_name()));
3304     mark_persistent(g_bool_true->raw());
3305 }
3306
3307 void finalize_bool() {
3308     delete g_bool;
3309     delete g_bool_false;
3310     delete g_bool_true;
3311 }
3312
3313 expr mk_bool() { return *g_bool; }
3314 expr mk_bool_true() { return *g_bool_true; }
3315 expr mk_bool_false() { return *g_bool_false; }
3316 expr to_bool_expr(bool b) { return b ? mk_bool_true() : mk_bool_false(); }
3317
3318 name get_dep_recursor(environment const &, name const &n) {
3319     return name(n, g_rec);
3320 }
3321
3322 name get_dep_cases_on(environment const &, name const &n) {
3323     return name(n, g_cases_on);
3324 }
3325
3326 extern "C" object *lean_mk_unsafe_rec_name(object *);
3327 extern "C" object *lean_is_unsafe_rec_name(object *);
3328
3329 name mk_unsafe_rec_name(name const &n) {
3330     return name(lean_mk_unsafe_rec_name(n.to_obj_arg()));
3331 }
3332
3333 optional<name> is_unsafe_rec_name(name const &n) {
3334     return option_ref<name>(lean_is_unsafe_rec_name(n.to_obj_arg())).get();
3335 }
3336
3337 static std::string *g_version_string = nullptr;
3338 std::string const &get_version_string() { return *g_version_string; }
3339
3340 expr const &extract_mdata(expr const &e) {
3341     if (is_mdata(e)) {
3342         return extract_mdata(mdata_expr(e));
3343     } else {
3344         return e;
3345     }
3346 }
3347
3348 optional<expr> to_optional_expr(obj_arg o) {
3349     if (is_scalar(o)) return none_expr();
3350     optional<expr> r = some_expr(expr(cnstr_get(o, 0), true));
3351     dec(o);
3352     return r;
3353 }
3354
3355 void initialize_library_util() {
3356     g_unit = new expr(mk_constant(get_unit_name()));
3357     mark_persistent(g_unit->raw());
3358     g_unit_mk = new expr(mk_constant(get_unit_unit_name()));
3359     mark_persistent(g_unit_mk->raw());
```

```
3360     g_true = new expr(mk_constant(get_true_name()));
3361     mark_persistent(g_true->raw());
3362     g_true_intro = new expr(mk_constant(get_true_intro_name()));
3363     mark_persistent(g_true_intro->raw());
3364     g_and = new expr(mk_constant(get_and_name()));
3365     mark_persistent(g_and->raw());
3366     g_and_intro = new expr(mk_constant(get_and_intro_name()));
3367     mark_persistent(g_and_intro->raw());
3368     g_and_left = new expr(mk_constant(get_and_left_name()));
3369     mark_persistent(g_and_left->raw());
3370     g_and_right = new expr(mk_constant(get_and_right_name()));
3371     mark_persistent(g_and_right->raw());
3372     initialize_nat();
3373     initialize_int();
3374     initialize_char();
3375     initialize_bool();
3376
3377     sstream out;
3378
3379     out << LEAN_VERSION_MAJOR << "." << LEAN_VERSION_MINOR << "."
3380         << LEAN_VERSION_PATCH;
3381     if (std::strlen(LEAN_SPECIAL_VERSION_DESC) > 0) {
3382         out << "-" << LEAN_SPECIAL_VERSION_DESC;
3383     }
3384     if (std::strcmp(LEAN_GITHASH, "GITDIR-NOTFOUND") == 0) {
3385         if (std::strcmp(LEAN_PACKAGE_VERSION, "NOT-FOUND") != 0) {
3386             out << ", package " << LEAN_PACKAGE_VERSION;
3387         }
3388     } else {
3389         out << ", commit " << std::string(LEAN_GITHASH).substr(0, 12);
3390     }
3391     g_version_string = new std::string(out.str());
3392
3393     g_util_fresh = new name("_util_fresh");
3394     mark_persistent(g_util_fresh->raw());
3395     register_name_generator_prefix(*g_util_fresh);
3396 }
3397
3398 void finalize_library_util() {
3399     delete g_util_fresh;
3400     delete g_version_string;
3401     finalize_bool();
3402     finalize_int();
3403     finalize_nat();
3404     finalize_char();
3405     delete g_true;
3406     delete g_true_intro;
3407     delete g_and;
3408     delete g_and_intro;
3409     delete g_and_left;
3410     delete g_and_right;
3411     delete g_unit_mk;
3412     delete g_unit;
3413 }
3414 }  // namespace lean
3415 // ::::::::::::::
3416 // compiler/borrowed_annotation.cpp
3417 // ::::::::::::::
3418 /*
3419 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
3420 Released under Apache 2.0 license as described in the file LICENSE.
3421
3422 Author: Leonardo de Moura
3423 */
3424 #include "kernel/instantiate.h"
3425 #include "library/annotation.h"
3426 #include "library/compiler/export_attribute.h"
3427 #include "library/compiler/extern_attribute.h"
3428 #include "library/compiler/llnf.h"
3429 #include "library/compiler/util.h"
```

```
3430 #include "library/trace.h"
3431
3432 namespace lean {
3433 static name *g_borrowed = nullptr;
3434
3435 expr mk_borrowed(expr const &e) { return mk_annotation(*g_borrowed, e); }
3436
3437 /*
3438 The new and old frontend use different approaches more annotating expressions.
3439 We use the following hacks to make sure we recognize both of them at
3440 `is_borrowed`.
3441 */
3442 extern "C" uint8 lean_is_marked_borrowed(lean_object *o);
3443
3444 bool is_borrowed(expr const &e) {
3445     expr e2 = e;
3446     return is_annotation(e2, *g_borrowed) ||
3447            lean_is_marked_borrowed(e2.to_obj_arg());
3448 }
3449 expr get_borrowed_arg(expr const &e) {
3450     lean_assert(is_borrowed(e));
3451     expr e2 = e;
3452     return mdata_expr(e2);
3453 }
3454
3455 void initialize_borrowed_annotation() {
3456     g_borrowed = new name("borrowed");
3457     mark_persistent(g_borrowed->raw());
3458     register_annotation(*g_borrowed);
3459 }
3460
3461 void finalize_borrowed_annotation() { delete g_borrowed; }
3462 }  // namespace lean
3463 // ::::::::::::::
3464 // compiler/closed_term_cache.cpp
3465 // ::::::::::::::
3466 /*
3467 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
3468 Released under Apache 2.0 license as described in the file LICENSE.
3469
3470 Author: Leonardo de Moura
3471 */
3472 #include "library/util.h"
3473
3474 namespace lean {
3475 extern "C" object *lean_cache_closed_term_name(object *env, object *e,
3476                                                object *n);
3477 extern "C" object *lean_get_closed_term_name(object *env, object *e);
3478
3479 optional<name> get_closed_term_name(environment const &env, expr const &e) {
3480     return to_optional<name>(
3481         lean_get_closed_term_name(env.to_obj_arg(), e.to_obj_arg()));
3482 }
3483
3484 environment cache_closed_term_name(environment const &env, expr const &e,
3485                                    name const &n) {
3486     return environment(lean_cache_closed_term_name(
3487         env.to_obj_arg(), e.to_obj_arg(), n.to_obj_arg()));
3488 }
3489 }  // namespace lean
3490 // ::::::::::::::
3491 // compiler/compiler.cpp
3492 // ::::::::::::::
3493 /*
3494 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
3495 Released under Apache 2.0 license as described in the file LICENSE.
3496
3497 Author: Leonardo de Moura
3498 */
3499 #include "kernel/kernel_exception.h"
```

```
3500 #include "kernel/type_checker.h"
3501 #include "library/compiler/cse.h"
3502 #include "library/compiler/csimp.h"
3503 #include "library/compiler/eager_lambda_lifting.h"
3504 #include "library/compiler/elim_dead_let.h"
3505 #include "library/compiler/erase_irrelevant.h"
3506 #include "library/compiler/export_attribute.h"
3507 #include "library/compiler/extern_attribute.h"
3508 #include "library/compiler/extract_closed.h"
3509 #include "library/compiler/find_jp.h"
3510 #include "library/compiler/ir.h"
3511 #include "library/compiler/lambda_lifting.h"
3512 #include "library/compiler/lcnf.h"
3513 #include "library/compiler/ll_infer_type.h"
3514 #include "library/compiler/llnf.h"
3515 #include "library/compiler/reduce_arity.h"
3516 #include "library/compiler/simp_app_args.h"
3517 #include "library/compiler/specialize.h"
3518 #include "library/compiler/struct_cases_on.h"
3519 #include "library/compiler/util.h"
3520 #include "library/max_sharing.h"
3521 #include "library/time_task.h"
3522 #include "library/trace.h"
3523 #include "util/io.h"
3524 #include "util/option_declarations.h"
3525
3526 namespace lean {
3527 static name *g_codegen = nullptr;
3528 static name *g_extract_closed = nullptr;
3529
3530 bool is_codegen_enabled(options const &opts) {
3531     return opts.get_bool(*g_codegen, true);
3532 }
3533 bool is_extract_closed_enabled(options const &opts) {
3534     return opts.get_bool(*g_extract_closed, true);
3535 }
3536
3537 static name get_real_name(name const &n) {
3538     if (optional<name> new_n = is_unsafe_rec_name(n))
3539         return *new_n;
3540     else
3541         return n;
3542 }
3543
3544 static comp_decls to_comp_decls(environment const &env, names const &cs) {
3545     bool allow_opaque = true;
3546     return map2<comp_decl>(cs, [&](name const &n) {
3547         return comp_decl(get_real_name(n), env.get(n).get_value(allow_opaque));
3548     });
3549 }
3550
3551 static expr eta_expand(environment const &env, expr const &e) {
3552     return type_checker(env).eta_expand(e);
3553 }
3554
3555 template <typename F>
3556 comp_decls apply(F &&f, environment const &env, comp_decls const &ds) {
3557     return map(ds, [&](comp_decl const &d) {
3558         return comp_decl(d.fst(), f(env, d.snd()));
3559     });
3560 }
3561
3562 template <typename F>
3563 comp_decls apply(F &&f, comp_decls const &ds) {
3564     return map(
3565         ds, [&](comp_decl const &d) { return comp_decl(d.fst(), f(d.snd())); });
3566 }
3567
3568 void trace_comp_decl(comp_decl const &d) {
3569     tout() << ">> " << d.fst() << "\n" << trace_pp_expr(d.snd()) << "\n";
```

```
}

void trace_comp_decls(comp_decls const &ds) {
    for (comp_decl const &d : ds) {
        trace_comp_decl(d);
    }
}

static environment cache_stage1(environment env, comp_decls const &ds) {
    for (comp_decl const &d : ds) {
        name n = d.fst();
        expr v = d.snd();
        constant_info info = env.get(n);
        env = register_stage1_decl(env, n, info.get_lparams(), info.get_type(),
                                   v);
    }
    return env;
}

static expr ensure_arity(expr const &t, unsigned arity) {
    if (arity == 0) {
        if (is_pi(t))
            return mk_enf_object_type();  // closure
        else
            return t;
    }
    lean_assert(is_pi(t));
    return update_binding(t, binding_domain(t),
                          ensure_arity(binding_body(t), arity - 1));
}

static environment cache_stage2(environment env, comp_decls const &ds,
                                bool only_new_ones = false) {
    buffer<expr> ts;
    ll_infer_type(env, ds, ts);
    lean_assert(ts.size() == length(ds));
    unsigned i = 0;
    for (comp_decl const &d : ds) {
        name n = d.fst();
        expr v = d.snd();
        if (!only_new_ones || !is_stage2_decl(env, n)) {
            expr t = ts[i];
            unsigned arity = get_num_nested_lambdas(v);
            t = ensure_arity(t, arity);
            lean_trace(name({"compiler", "stage2"}),
                       tout() << n << " : " << t << "\n";);
            lean_trace(name({"compiler", "ll_infer_type"}),
                       tout() << n << " : " << t << "\n";);
            env = register_stage2_decl(env, n, t, v);
        }
        i++;
    }
    return env;
}

/* Cache the declarations in `ds` that have not already been cached. */
static environment cache_new_stage2(environment env, comp_decls const &ds) {
    return cache_stage2(env, ds, true);
}

bool is_main_fn(environment const &env, name const &n) {
    if (n == "main") return true;
    if (optional<name> c = get_export_name_for(env, n)) {
        return *c == "main";
    }
    return false;
}

bool is_uint32_or_unit(expr const &type) {
    return is_constant(type, get_uint32_name()) ||
```

```
3640            is_constant(type, get_unit_name()) ||
3641            is_constant(type, get_punit_name());
3642 }
3643
3644 /* Return true iff type is `List String -> IO UInt32` or `IO UInt32` */
3645 bool is_main_fn_type(expr const &type) {
3646     if (is_arrow(type)) {
3647         expr d = binding_domain(type);
3648         expr r = binding_body(type);
3649         return is_app(r) && is_constant(app_fn(r), get_io_name()) &&
3650                is_uint32_or_unit(app_arg(r)) && is_app(d) &&
3651                is_constant(app_fn(d), get_list_name()) &&
3652                is_constant(app_arg(d), get_string_name());
3653     } else if (is_app(type)) {
3654         return is_constant(app_fn(type), get_io_name()) &&
3655                is_uint32_or_unit(app_arg(type));
3656     } else {
3657         return false;
3658     }
3659 }
3660
3661 #define trace_compiler(k, ds) lean_trace(k, trace_comp_decls(ds););
3662
3663 environment compile(environment const &env, options const &opts, names cs) {
3664     if (!is_codegen_enabled(opts)) return env;
3665
3666     /* Do not generate code for irrelevant decls */
3667     cs = filter(cs, [&](name const &c) {
3668         return !is_irrelevant_type(env, env.get(c).get_type());
3669     });
3670     if (empty(cs)) return env;
3671
3672     for (name const &c : cs) {
3673         if (is_main_fn(env, c) && !is_main_fn_type(env.get(c).get_type())) {
3674             throw exception(
3675                 "invalid `main` function, it must have type `List String -> IO "
3676                 "UInt32`");
3677         }
3678     }
3679
3680     if (length(cs) == 1) {
3681         name c = get_real_name(head(cs));
3682         if (is_extern_constant(env, c)) {
3683             /* Generate boxed version for extern/native constant if needed. */
3684             return ir::add_extern(env, c);
3685         }
3686     }
3687
3688     for (name const &c : cs) {
3689         lean_assert(!is_extern_constant(env, get_real_name(c)));
3690         constant_info cinfo = env.get(c);
3691         if (!cinfo.is_definition() && !cinfo.is_opaque()) return env;
3692     }
3693
3694     time_task t("compilation", opts);
3695     scope_trace_env scope_trace(env, opts);
3696
3697     comp_decls ds = to_comp_decls(env, cs);
3698     csimp_cfg cfg(opts);
3699     // Use the following line to see compiler intermediate steps
3700     // scope_traces_as_string trace_scope;
3701     auto simp = [&](environment const &env, expr const &e) {
3702         return csimp(env, e, cfg);
3703     };
3704     auto esimp = [&](environment const &env, expr const &e) {
3705         return cesimp(env, e, cfg);
3706     };
3707     trace_compiler(name({"compiler", "input"}), ds);
3708     ds = apply(eta_expand, env, ds);
3709     trace_compiler(name({"compiler", "eta_expand"}), ds);
```

```
3710     ds = apply(to_lcnf, env, ds);
3711     ds = apply(find_jp, env, ds);
3712     // trace(ds);
3713     trace_compiler(name({"compiler", "lcnf"}), ds);
3714     // trace(ds);
3715     ds = apply(cce, env, ds);
3716     trace_compiler(name({"compiler", "cce"}), ds);
3717     ds = apply(simp, env, ds);
3718     trace_compiler(name({"compiler", "simp"}), ds);
3719     // trace(ds);
3720     environment new_env = env;
3721     std::tie(new_env, ds) = eager_lambda_lifting(new_env, ds, cfg);
3722     trace_compiler(name({"compiler", "eager_lambda_lifting"}), ds);
3723     ds = apply(max_sharing, ds);
3724     trace_compiler(name({"compiler", "stage1"}), ds);
3725     new_env = cache_stage1(new_env, ds);
3726     std::tie(new_env, ds) = specialize(new_env, ds, cfg);
3727     lean_assert(lcnf_check_let_decls(new_env, ds));
3728     trace_compiler(name({"compiler", "specialize"}), ds);
3729     ds = apply(elim_dead_let, ds);
3730     trace_compiler(name({"compiler", "elim_dead_let"}), ds);
3731     ds = apply(erase_irrelevant, new_env, ds);
3732     trace_compiler(name({"compiler", "erase_irrelevant"}), ds);
3733     ds = apply(struct_cases_on, new_env, ds);
3734     trace_compiler(name({"compiler", "struct_cases_on"}), ds);
3735     ds = apply(esimp, new_env, ds);
3736     trace_compiler(name({"compiler", "simp"}), ds);
3737     ds = reduce_arity(new_env, ds);
3738     trace_compiler(name({"compiler", "reduce_arity"}), ds);
3739     std::tie(new_env, ds) = lambda_lifting(new_env, ds);
3740     trace_compiler(name({"compiler", "lambda_lifting"}), ds);
3741     // trace(ds);
3742     ds = apply(esimp, new_env, ds);
3743     trace_compiler(name({"compiler", "simp"}), ds);
3744     new_env = cache_stage2(new_env, ds);
3745     trace_compiler(name({"compiler", "stage2"}), ds);
3746     if (is_extract_closed_enabled(opts)) {
3747         std::tie(new_env, ds) = extract_closed(new_env, ds);
3748         ds = apply(elim_dead_let, ds);
3749         ds = apply(esimp, new_env, ds);
3750         trace_compiler(name({"compiler", "extract_closed"}), ds);
3751     }
3752     new_env = cache_new_stage2(new_env, ds);
3753     ds = apply(esimp, new_env, ds);
3754     trace_compiler(name({"compiler", "simp"}), ds);
3755     ds = apply(simp_app_args, new_env, ds);
3756     ds = apply(ecse, new_env, ds);
3757     ds = apply(elim_dead_let, ds);
3758     trace_compiler(name({"compiler", "simp_app_args"}), ds);
3759     // std::cout << trace_scope.get_string() << "\n";
3760     /* compile IR. */
3761     return compile_ir(new_env, opts, ds);
3762 }
3763
3764 extern "C" object *lean_get_decl_names_for_code_gen(object *);
3765 names get_decl_names_for_code_gen(declaration const &decl) {
3766     return names(lean_get_decl_names_for_code_gen(decl.to_obj_arg()));
3767 }
3768
3769 extern "C" object *lean_compile_decl(object *env, object *opts, object *decl) {
3770     return catch_kernel_exceptions<environment>([&]() {
3771         return compile(environment(env), options(opts, true),
3772                        get_decl_names_for_code_gen(declaration(decl, true)));
3773     });
3774 }
3775
3776 void initialize_compiler() {
3777     g_codegen = new name("codegen");
3778     mark_persistent(g_codegen->raw());
3779     g_extract_closed = new name{"compiler", "extract_closed"};
```

```
3780     mark_persistent(g_extract_closed->raw());
3781     register_bool_option(*g_codegen, true,
3782                          "(compiler) enable/disable code generation");
3783     register_bool_option(*g_extract_closed, true,
3784                          "(compiler) enable/disable closed term caching");
3785     register_trace_class("compiler");
3786     register_trace_class({"compiler", "input"});
3787     register_trace_class({"compiler", "eta_expand"});
3788     register_trace_class({"compiler", "lcnf"});
3789     register_trace_class({"compiler", "cce"});
3790     register_trace_class({"compiler", "simp"});
3791     register_trace_class({"compiler", "simp_detail"});
3792     register_trace_class({"compiler", "simp_float_cases"});
3793     register_trace_class({"compiler", "elim_dead_let"});
3794     register_trace_class({"compiler", "cse"});
3795     register_trace_class({"compiler", "specialize"});
3796     register_trace_class({"compiler", "stage1"});
3797     register_trace_class({"compiler", "stage2"});
3798     register_trace_class({"compiler", "erase_irrelevant"});
3799     register_trace_class({"compiler", "eager_lambda_lifting"});
3800     register_trace_class({"compiler", "lambda_lifting"});
3801     register_trace_class({"compiler", "extract_closed"});
3802     register_trace_class({"compiler", "reduce_arity"});
3803     register_trace_class({"compiler", "simp_app_args"});
3804     register_trace_class({"compiler", "struct_cases_on"});
3805     register_trace_class({"compiler", "llnf"});
3806     register_trace_class({"compiler", "result"});
3807     register_trace_class({"compiler", "optimize_bytecode"});
3808     register_trace_class({"compiler", "code_gen"});
3809     register_trace_class({"compiler", "ll_infer_type"});
3810     register_trace_class({"compiler", "ir"});
3811     register_trace_class({"compiler", "ir", "init"});
3812     register_trace_class({"compiler", "ir", "push_proj"});
3813     register_trace_class({"compiler", "ir", "reset_reuse"});
3814     register_trace_class({"compiler", "ir", "elim_dead_branches"});
3815     register_trace_class({"compiler", "ir", "elim_dead"});
3816     register_trace_class({"compiler", "ir", "simp_case"});
3817     register_trace_class({"compiler", "ir", "borrow"});
3818     register_trace_class({"compiler", "ir", "boxing"});
3819     register_trace_class({"compiler", "ir", "rc"});
3820     register_trace_class({"compiler", "ir", "expand_reset_reuse"});
3821     register_trace_class({"compiler", "ir", "result"});
3822 }
3823
3824 void finalize_compiler() {
3825     delete g_codegen;
3826     delete g_extract_closed;
3827 }
3828 }  // namespace lean
3829 // ::::::::::::::
3830 // compiler/cse.cpp
3831 // ::::::::::::::
3832 /*
3833 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
3834 Released under Apache 2.0 license as described in the file LICENSE.
3835
3836 Author: Leonardo de Moura
3837 */
3838 #include <lean/flet.h>
3839
3840 #include <algorithm>
3841 #include <vector>
3842
3843 #include "kernel/abstract.h"
3844 #include "kernel/environment.h"
3845 #include "kernel/expr_maps.h"
3846 #include "kernel/expr_sets.h"
3847 #include "kernel/for_each_fn.h"
3848 #include "kernel/instantiate.h"
3849 #include "kernel/replace_fn.h"
```

```
#include "library/compiler/util.h"
#include "util/name_generator.h"

namespace lean {
static name *g_cse_fresh = nullptr;

class cse_fn {
    environment m_env;
    name_generator m_ngen;
    bool m_before_erasure;
    expr_map<expr> m_map;
    std::vector<expr> m_keys;

   public:
    expr mk_key(expr const &type, expr const &val) {
        if (m_before_erasure) {
            return val;
        } else {
            /* After erasure, we should also compare the type. For example, we
               might have

                  x_1 : uint32 := 0
                  x_2 : uint8  := 0

               which are different at runtime. We might also have

                  x_1 : uint8 := _cnstr.0.0.0
                  x_2 : _obj  := _cnstr.0.0.0

               where x_1 is representing a value of an enumeration type,
               and x_2 list.nil.

               We encode the pair using an application.
               This solution is a bit hackish, and we should try to refine it in
               the future. */
            return mk_app(type, val);
        }
    }

    bool has_never_extract(expr const &e) {
        expr const &fn = get_app_fn(e);
        return is_constant(fn) &&
               has_never_extract_attribute(m_env, const_name(fn));
    }

    expr visit_let(expr e) {
        unsigned keys_size = m_keys.size();
        buffer<expr> fvars;
        buffer<expr> to_keep_fvars;
        buffer<std::tuple<name, expr, expr>> entries;
        while (is_let(e)) {
            expr value =
                instantiate_rev(let_value(e), fvars.size(), fvars.data());
            expr type =
                instantiate_rev(let_type(e), fvars.size(), fvars.data());
            expr key = mk_key(type, value);
            auto it = m_map.find(key);
            if (it != m_map.end()) {
                lean_assert(is_fvar(it->second));
                fvars.push_back(it->second);
            } else {
                expr new_value = visit(value);
                expr fvar = mk_fvar(m_ngen.next());
                fvars.push_back(fvar);
                to_keep_fvars.push_back(fvar);
                entries.emplace_back(let_name(e), type, new_value);
                if (!is_cases_on_app(m_env, new_value) &&
                    !has_never_extract(new_value)) {
                    expr new_key = mk_key(type, new_value);
                    m_map.insert(mk_pair(new_key, fvar));
```

```
                    m_keys.push_back(new_key);
                }
            }
            e = let_body(e);
        }
        e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
        e = abstract(e, to_keep_fvars.size(), to_keep_fvars.data());
        lean_assert(entries.size() == to_keep_fvars.size());
        unsigned i = entries.size();
        while (i > 0) {
            --i;
            expr new_type =
                abstract(std::get<1>(entries[i]), i, to_keep_fvars.data());
            expr new_value =
                abstract(std::get<2>(entries[i]), i, to_keep_fvars.data());
            e = mk_let(std::get<0>(entries[i]), new_type, new_value, e);
        }
        /* Restore m_map */
        for (unsigned i = keys_size; i < m_keys.size(); i++) {
            m_map.erase(m_keys[i]);
        }
        m_keys.resize(keys_size);
        return e;
    }

    expr visit_lambda(expr e) {
        buffer<expr> fvars;
        buffer<std::tuple<name, expr, binder_info>> entries;
        while (is_lambda(e)) {
            expr domain =
                instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
            expr fvar = mk_fvar(m_ngen.next());
            entries.emplace_back(binding_name(e), domain, binding_info(e));
            fvars.push_back(fvar);
            e = binding_body(e);
        }
        e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
        e = abstract(e, fvars.size(), fvars.data());
        unsigned i = entries.size();
        while (i > 0) {
            --i;
            expr new_domain =
                abstract(std::get<1>(entries[i]), i, fvars.data());
            e = mk_lambda(std::get<0>(entries[i]), new_domain, e,
                          std::get<2>(entries[i]));
        }
        return e;
    }

    expr visit_app(expr const &e) {
        if (is_cases_on_app(m_env, e)) {
            buffer<expr> args;
            expr const &c = get_app_args(e, args);
            lean_assert(is_constant(c));
            unsigned minor_idx;
            unsigned minors_end;
            std::tie(minor_idx, minors_end) = get_cases_on_minors_range(
                m_env, const_name(c), m_before_erasure);
            for (unsigned i = minor_idx; i < minors_end; i++) {
                args[i] = visit(args[i]);
            }
            return mk_app(c, args);
        } else {
            return e;
        }
    }

    expr visit(expr const &e) {
        switch (e.kind()) {
            case expr_kind::Lambda:
```

```
                return visit_lambda(e);
            case expr_kind::App:
                return visit_app(e);
            case expr_kind::Let:
                return visit_let(e);
            default:
                return e;
        }
    }

   public:
    cse_fn(environment const &env, bool before_erasure)
        : m_env(env), m_ngen(*g_cse_fresh), m_before_erasure(before_erasure) {}

    expr operator()(expr const &e) { return visit(e); }
};

expr cse_core(environment const &env, expr const &e, bool before_erasure) {
    return cse_fn(env, before_erasure)(e);
}

/* Common case elimination.

   This transformation creates join-points for identical minor premises.
   This is important in code such as
   ```
   def get_fn : expr -> tactic expr
   | (expr.app f _) := pure f
   | _   := throw "expr is not an application"
   ```
   The "else"-branch is duplicated by the equation compiler for each constructor
   different from `expr.app`. */
class cce_fn {
    type_checker::state m_st;
    local_ctx m_lctx;
    buffer<expr> m_fvars;
    expr_map<bool> m_cce_candidates;
    buffer<expr> m_cce_targets;
    name m_j;
    unsigned m_next_idx{1};

   public:
    environment &env() { return m_st.env(); }

    name_generator &ngen() { return m_st.ngen(); }

    unsigned get_fvar_idx(expr const &x) {
        return m_lctx.get_local_decl(x).get_idx();
    }

    unsigned get_max_fvar_idx(expr const &e) {
        if (!has_fvar(e)) return 0;
        unsigned r = 0;
        for_each(e, [&](expr const &x, unsigned) {
            if (!has_fvar(x)) return false;
            if (is_fvar(x)) {
                unsigned x_idx = get_fvar_idx(x);
                if (x_idx > r) r = x_idx;
            }
            return true;
        });
        return r;
    }

    expr replace_target(expr const &e, expr const &target, expr const &jmp) {
        return replace(e, [&](expr const &t, unsigned) {
            if (target == t) {
                return some_expr(jmp);
            }
            return none_expr();
```

```
4060        });
4061    }
4062
4063    expr mk_let_lambda(unsigned old_fvars_size, expr body, bool is_let) {
4064        lean_assert(m_fvars.size() >= old_fvars_size);
4065        if (m_fvars.size() == old_fvars_size) return body;
4066        unsigned first_var_idx;
4067        if (old_fvars_size == 0)
4068            first_var_idx = 0;
4069        else
4070            first_var_idx = get_fvar_idx(m_fvars[old_fvars_size]);
4071        unsigned j = 0;
4072        buffer<pair<expr, expr>> target_jmp_pairs;
4073        name_set new_fvar_names;
4074        for (unsigned i = 0; i < m_cce_targets.size(); i++) {
4075            expr target = m_cce_targets[i];
4076            unsigned max_idx = get_max_fvar_idx(target);
4077            if (max_idx >= first_var_idx) {
4078                expr target_type =
4079                    cheap_beta_reduce(type_checker(m_st, m_lctx).infer(target));
4080                expr unit = mk_unit();
4081                expr unit_mk = mk_unit_mk();
4082                expr target_val = target;
4083                if (is_lambda(target_val)) {
4084                    /* Make sure we don't change the arity of the joint point.
4085                       We use a "trivial let" to encode a joint point that
4086                       returns a lambda:
4087                       ```
4088                          jp : unit -> target_type :=
4089                          fun _ : unit, let _x : target_type := target_val in _x
4090                       ```
4091                    */
4092                    target_val = ::lean::mk_let("_x", target_type, target_val,
4093                                                mk_bvar(0));
4094                }
4095                expr new_val = ::lean::mk_lambda("u", unit, target_val);
4096                expr new_type = ::lean::mk_arrow(unit, target_type);
4097                expr new_fvar = m_lctx.mk_local_decl(
4098                    ngen(), mk_join_point_name(m_j.append_after(m_next_idx)),
4099                    new_type, new_val);
4100                new_fvar_names.insert(fvar_name(new_fvar));
4101                expr jmp = mk_app(new_fvar, unit_mk);
4102                if (is_let) {
4103                    /* We must insert new_fvar after fvar with idx == max_idx */
4104                    m_next_idx++;
4105                    unsigned k = old_fvars_size;
4106                    for (; k < m_fvars.size(); k++) {
4107                        expr const &fvar = m_fvars[k];
4108                        if (get_fvar_idx(fvar) > max_idx) {
4109                            m_fvars.insert(k, new_fvar);
4110                            /* We need to save the pairs to replace the `target`
4111                             * on let-declarations that occurr after k */
4112                            target_jmp_pairs.emplace_back(target, jmp);
4113                            break;
4114                        }
4115                    }
4116                    if (k == m_fvars.size()) {
4117                        m_fvars.push_back(new_fvar);
4118                    }
4119                } else {
4120                    lean_assert(!is_let);
4121                    /* For lambda we add new free variable after lambda vars */
4122                    m_fvars.push_back(new_fvar);
4123                }
4124                body = replace_target(body, target, jmp);
4125            } else {
4126                m_cce_targets[j] = target;
4127                j++;
4128            }
4129        }
```

```
4130         m_cce_targets.shrink(j);
4131         if (is_let && !target_jmp_pairs.empty()) {
4132             expr r = abstract(body, m_fvars.size() - old_fvars_size,
4133                               m_fvars.data() + old_fvars_size);
4134             unsigned i = m_fvars.size();
4135             while (i > old_fvars_size) {
4136                 --i;
4137                 expr fvar = m_fvars[i];
4138                 local_decl decl = m_lctx.get_local_decl(fvar);
4139                 expr type = abstract(decl.get_type(), i - old_fvars_size,
4140                                      m_fvars.data() + old_fvars_size);
4141                 lean_assert(decl.get_value());
4142                 expr val = *decl.get_value();
4143                 if ((!new_fvar_names.contains(fvar_name(fvar))) &&
4144                     (is_lambda(val) || is_cases_on_app(env(), val))) {
4145                     for (pair<expr, expr> const &p : target_jmp_pairs) {
4146                         val = replace_target(val, p.first, p.second);
4147                     }
4148                 }
4149                 val = abstract(val, i - old_fvars_size,
4150                                m_fvars.data() + old_fvars_size);
4151                 r = ::lean::mk_let(decl.get_user_name(), type, val, r);
4152             }
4153             m_fvars.shrink(old_fvars_size);
4154             return r;
4155         } else {
4156             expr r = m_lctx.mk_lambda(m_fvars.size() - old_fvars_size,
4157                                       m_fvars.data() + old_fvars_size, body);
4158             m_fvars.shrink(old_fvars_size);
4159             return r;
4160         }
4161     }
4162
4163     expr mk_let(unsigned old_fvars_size, expr const &body) {
4164         return mk_let_lambda(old_fvars_size, body, true);
4165     }
4166
4167     expr mk_lambda(unsigned old_fvars_size, expr const &body) {
4168         return mk_let_lambda(old_fvars_size, body, false);
4169     }
4170
4171     expr visit_let(expr e) {
4172         buffer<expr> let_fvars;
4173         while (is_let(e)) {
4174             expr new_type = instantiate_rev(let_type(e), let_fvars.size(),
4175                                             let_fvars.data());
4176             expr new_val = visit(instantiate_rev(let_value(e), let_fvars.size(),
4177                                                  let_fvars.data()));
4178             expr new_fvar =
4179                 m_lctx.mk_local_decl(ngen(), let_name(e), new_type, new_val);
4180             let_fvars.push_back(new_fvar);
4181             m_fvars.push_back(new_fvar);
4182             e = let_body(e);
4183         }
4184         return instantiate_rev(e, let_fvars.size(), let_fvars.data());
4185     }
4186
4187     expr visit_lambda(expr e) {
4188         lean_assert(is_lambda(e));
4189         flet<local_ctx> save_lctx(m_lctx, m_lctx);
4190         unsigned fvars_sz1 = m_fvars.size();
4191         while (is_lambda(e)) {
4192             /* Types are ignored in compilation steps. So, we do not invoke
4193              * visit for d. */
4194             expr new_d =
4195                 instantiate_rev(binding_domain(e), m_fvars.size() - fvars_sz1,
4196                                 m_fvars.data() + fvars_sz1);
4197             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e), new_d,
4198                                                  binding_info(e));
4199             m_fvars.push_back(new_fvar);
```

```
4200            e = binding_body(e);
4201        }
4202        unsigned fvars_sz2 = m_fvars.size();
4203        expr new_body = visit(instantiate_rev(e, m_fvars.size() - fvars_sz1,
4204                                              m_fvars.data() + fvars_sz1));
4205        new_body = mk_let(fvars_sz2, new_body);
4206        return mk_lambda(fvars_sz1, new_body);
4207    }
4208
4209    void add_candidate(expr const &e) {
4210        /* TODO(Leo): we should not consider `e` if it is small */
4211        auto it = m_cce_candidates.find(e);
4212        if (it == m_cce_candidates.end()) {
4213            m_cce_candidates.insert(mk_pair(e, true));
4214        } else if (it->second) {
4215            m_cce_targets.push_back(e);
4216            it->second = false;
4217        }
4218    }
4219
4220    expr visit_app(expr const &e) {
4221        if (!is_cases_on_app(env(), e)) return e;
4222        buffer<expr> args;
4223        expr const &c = get_app_args(e, args);
4224        lean_assert(is_constant(c));
4225        inductive_val I_val =
4226            env().get(const_name(c).get_prefix()).to_inductive_val();
4227        unsigned motive_idx = I_val.get_nparams();
4228        unsigned first_index = motive_idx + 1;
4229        unsigned nindices = I_val.get_nindices();
4230        unsigned major_idx = first_index + nindices;
4231        unsigned first_minor_idx = major_idx + 1;
4232        unsigned nminors = length(I_val.get_cnstrs());
4233        /* visit minor premises */
4234        for (unsigned i = 0; i < nminors; i++) {
4235            unsigned minor_idx = first_minor_idx + i;
4236            expr minor = args[minor_idx];
4237            flet<local_ctx> save_lctx(m_lctx, m_lctx);
4238            unsigned fvars_sz1 = m_fvars.size();
4239            while (is_lambda(minor)) {
4240                expr new_d = instantiate_rev(binding_domain(minor),
4241                                             m_fvars.size() - fvars_sz1,
4242                                             m_fvars.data() + fvars_sz1);
4243                expr new_fvar = m_lctx.mk_local_decl(
4244                    ngen(), binding_name(minor), new_d, binding_info(minor));
4245                m_fvars.push_back(new_fvar);
4246                minor = binding_body(minor);
4247            }
4248            bool is_cce_target = !has_loose_bvars(minor);
4249            unsigned fvars_sz2 = m_fvars.size();
4250            expr new_minor = visit(instantiate_rev(
4251                minor, m_fvars.size() - fvars_sz1, m_fvars.data() + fvars_sz1));
4252            new_minor = mk_let(fvars_sz2, new_minor);
4253            if (is_cce_target && !is_lcnf_atom(new_minor))
4254                add_candidate(new_minor);
4255            new_minor = mk_lambda(fvars_sz1, new_minor);
4256            args[minor_idx] = new_minor;
4257        }
4258        return mk_app(c, args);
4259    }
4260
4261    expr visit(expr const &e) {
4262        switch (e.kind()) {
4263            case expr_kind::Lambda:
4264                return visit_lambda(e);
4265            case expr_kind::App:
4266                return visit_app(e);
4267            case expr_kind::Let:
4268                return visit_let(e);
4269            default:
```

```
                return e;
            }
        }

    public:
        cce_fn(environment const &env, local_ctx const &lctx)
            : m_st(env), m_lctx(lctx), m_j("_j") {}

        expr operator()(expr const &e) {
            expr r = visit(e);
            return mk_let(0, r);
        }
};

expr cce_core(environment const &env, local_ctx const &lctx, expr const &e) {
    return cce_fn(env, lctx)(e);
}

void initialize_cse() {
    g_cse_fresh = new name("_cse_fresh");
    mark_persistent(g_cse_fresh->raw());
    register_name_generator_prefix(*g_cse_fresh);
}
void finalize_cse() { delete g_cse_fresh; }
}  // namespace lean
// ::::::::::::::
// compiler/csimp.cpp
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include <algorithm>
#include <unordered_map>
#include <unordered_set>

#include "kernel/abstract.h"
#include "kernel/find_fn.h"
#include "kernel/for_each_fn.h"
#include "kernel/inductive.h"
#include "kernel/instantiate.h"
#include "kernel/kernel_exception.h"
#include "kernel/type_checker.h"
#include "library/class.h"
#include "library/compiler/cse.h"
#include "library/compiler/csimp.h"
#include "library/compiler/elim_dead_let.h"
#include "library/compiler/extract_closed.h"
#include "library/compiler/init_attribute.h"
#include "library/compiler/reduce_arity.h"
#include "library/compiler/util.h"
#include "library/constants.h"
#include "library/expr_pair_maps.h"
#include "library/trace.h"
#include "library/util.h"

namespace lean {
csimp_cfg::csimp_cfg(options const &) : csimp_cfg() {}

csimp_cfg::csimp_cfg() {
    m_inline = true;
    m_inline_threshold = 1;
    m_float_cases_threshold = 20;
    m_inline_jp_threshold = 2;
}

```

```
4340 /*
4341 @[export lean_fold_un_op]
4342 def fold_un_op (before_erasure : bool) (f : expr) (a : expr) : option expr :=
4343 */
4344 extern "C" object *lean_fold_un_op(uint8 before_erasure, object *f, object *a);
4345
4346 optional<expr> fold_un_op(bool before_erasure, expr const &f, expr const &a) {
4347     inc(f.raw());
4348     inc(a.raw());
4349     return to_optional_expr(lean_fold_un_op(before_erasure, f.raw(), a.raw()));
4350 }
4351
4352 /*
4353 @[export lean_fold_bin_op]
4354 def fold_bin_op (before_erasure : bool) (f : expr) (a : expr) (b : expr) :
4355 option expr :=
4356 */
4357 extern "C" object *lean_fold_bin_op(uint8 before_erasure, object *f, object *a,
4358                                     object *b);
4359
4360 optional<expr> fold_bin_op(bool before_erasure, expr const &f, expr const &a,
4361                            expr const &b) {
4362     inc(f.raw());
4363     inc(a.raw());
4364     inc(b.raw());
4365     return to_optional_expr(
4366         lean_fold_bin_op(before_erasure, f.raw(), a.raw(), b.raw()));
4367 }
4368
4369 class csimp_fn {
4370     typedef expr_pair_struct_map<expr> jp_cache;
4371     type_checker::state m_st;
4372     local_ctx m_lctx;
4373     bool m_before_erasure;
4374     csimp_cfg m_cfg;
4375     buffer<expr> m_fvars;
4376     name m_x;
4377     name m_j;
4378     unsigned m_next_idx{1};
4379     unsigned m_next_jp_idx{1};
4380     expr_set m_simplified;
4381     /* Cache for the method `mk_new_join_point`. It maps the pair `(jp,
4382      * lambda(x, e))` to the new joint point. */
4383     jp_cache m_jp_cache;
4384     /* Maps a free variables to a list of joint points that must be inserted
4385      * after it. */
4386     expr_map<exprs> m_fvar2jps;
4387     /* Maps a new join point to the free variable it must be defined after.
4388        It is the "inverse" of m_fvar2jps. It maps to `none` if the joint point
4389        is in `m_closed_jps` */
4390     expr_map<optional<expr>> m_jp2fvar;
4391     /* Join points that do not depend on any free variable. */
4392     exprs m_closed_jps;
4393     /* Mapping from `casesOn` scrutinee to constructor it is bound to.
4394        We update the mapping when visiting a `cases_on` branch.
4395        For example, given
4396        ```
4397        List.cases_on x
4398          <nil_case>
4399          (fun h t, <cons_case h t>)
4400        ```
4401        We can assume `x` is bound to `h::t` when visiting `<cons_case h t>`.
4402        We use this information to reduce nested cases_on applications and
4403        projections. */
4404     typedef rb_expr_map<expr> expr2ctor;
4405     expr2ctor m_expr2ctor;
4406
4407     environment const &env() const { return m_st.env(); }
4408
4409     name_generator &ngen() { return m_st.ngen(); }
```

```
4410
4411    unsigned get_fvar_idx(expr const &x) {
4412        lean_assert(is_fvar(x));
4413        return m_lctx.get_local_decl(x).get_idx();
4414    }
4415
4416    optional<expr> find_max_fvar(expr const &e) {
4417        if (!has_fvar(e)) return none_expr();
4418        unsigned max_idx = 0;
4419        optional<expr> r;
4420        for_each(e, [&](expr const &x, unsigned) {
4421            if (!has_fvar(x)) return false;
4422            if (is_fvar(x)) {
4423                auto it = m_jp2fvar.find(x);
4424                expr y;
4425                if (it != m_jp2fvar.end()) {
4426                    if (!it->second) {
4427                        /* `x` is a join point in `m_closed_jps`. */
4428                        return false;
4429                    }
4430                    y = *it->second;
4431                } else {
4432                    y = x;
4433                }
4434                unsigned curr_idx = get_fvar_idx(y);
4435                if (!r || curr_idx > max_idx) {
4436                    r = y;
4437                    max_idx = curr_idx;
4438                }
4439            }
4440            return true;
4441        });
4442        return r;
4443    }
4444
4445    void register_new_jp(expr const &jp) {
4446        local_decl jp_decl = m_lctx.get_local_decl(jp);
4447        expr jp_val = *jp_decl.get_value();
4448        if (optional<expr> max_var = find_max_fvar(jp_val)) {
4449            m_jp2fvar.insert(mk_pair(jp, some_expr(*max_var)));
4450            auto it = m_fvar2jps.find(*max_var);
4451            if (it == m_fvar2jps.end()) {
4452                m_fvar2jps.insert(mk_pair(*max_var, exprs(jp)));
4453            } else {
4454                it->second = exprs(jp, it->second);
4455            }
4456        } else {
4457            m_jp2fvar.insert(mk_pair(jp, none_expr()));
4458            m_closed_jps = exprs(jp, m_closed_jps);
4459        }
4460    }
4461
4462    void check(expr const &e) {
4463        if (m_before_erasure) {
4464            try {
4465                type_checker(m_st, m_lctx).check(e);
4466            } catch (exception &) {
4467                lean_unreachable();
4468            }
4469        }
4470    }
4471
4472    void mark_simplified(expr const &e) { m_simplified.insert(e); }
4473
4474    bool already_simplified(expr const &e) const {
4475        return m_simplified.find(e) != m_simplified.end();
4476    }
4477
4478    bool is_join_point_app(expr const &e) const {
4479        if (!is_app(e)) return false;
```

```
4480         expr const &fn = get_app_fn(e);
4481         return is_fvar(fn) &&
4482                is_join_point_name(m_lctx.get_local_decl(fn).get_user_name());
4483     }
4484
4485     bool is_small_join_point(expr const &e) const {
4486         return get_lcnf_size(env(), e) <= m_cfg.m_inline_jp_threshold;
4487     }
4488
4489     expr find(expr const &e, bool skip_mdata = true,
4490               bool use_expr2ctor = false) const {
4491         if (use_expr2ctor) {
4492             if (expr const *ctor = m_expr2ctor.find(e)) {
4493                 return *ctor;
4494             }
4495         }
4496         if (is_fvar(e)) {
4497             if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
4498                 if (optional<expr> v = decl->get_value()) {
4499                     /* Pseudo "do" joinpoints are used to implement a temporary
4500                        HACK. See `visit_let` method at `lcnf.cpp`. Remark: the
4501                        condition `is_lambda(*v)` will be false after we perform
4502                        lambda lifting. */
4503                     if (!is_pseudo_do_join_point_name(decl->get_user_name()) ||
4504                         !is_lambda(*v)) {
4505                         if (!is_join_point_name(decl->get_user_name()))
4506                             return find(*v, skip_mdata, use_expr2ctor);
4507                         else if (is_small_join_point(*v))
4508                             return find(*v, skip_mdata, use_expr2ctor);
4509                     }
4510                 }
4511             }
4512         } else if (is_mdata(e) && skip_mdata) {
4513             return find(mdata_expr(e), true, use_expr2ctor);
4514         }
4515         return e;
4516     }
4517
4518     expr find_ctor(expr const &e) const { return find(e, true, true); }
4519
4520     type_checker tc() {
4521         lean_assert(m_before_erasure);
4522         return type_checker(m_st, m_lctx);
4523     }
4524
4525     expr infer_type(expr const &e) {
4526         if (m_before_erasure)
4527             return type_checker(m_st, m_lctx).infer(e);
4528         else
4529             return mk_enf_object_type();
4530     }
4531
4532     expr whnf(expr const &e) {
4533         lean_assert(m_before_erasure);
4534         return type_checker(m_st, m_lctx).whnf(e);
4535     }
4536
4537     expr whnf_infer_type(expr const &e) {
4538         lean_assert(m_before_erasure);
4539         type_checker tc(m_st, m_lctx);
4540         return tc.whnf(tc.infer(e));
4541     }
4542
4543     name next_name() {
4544         /* Remark: we use `m_x.append_after(m_next_idx)` instead of `name(m_x,
4545            m_next_idx)` because the resulting name is confusing during
4546            debugging: it looks like a projection application.
4547            We should replace it with `name(m_x, m_next_idx)` when the compiler
4548            code gets more stable. */
4549         name r = m_x.append_after(m_next_idx);
```

```
4550         m_next_idx++;
4551         return r;
4552     }
4553
4554     name next_jp_name() {
4555         name r = m_j.append_after(m_next_jp_idx);
4556         m_next_jp_idx++;
4557         return mk_join_point_name(r);
4558     }
4559
4560     /* Create a new let-declaration `x : t := e`, add `x` to `m_fvars` and
4561      * return `x`. */
4562     expr mk_let_decl(expr const &e) {
4563         lean_assert(!is_lcnf_atom(e));
4564         expr type = cheap_beta_reduce(infer_type(e));
4565         expr fvar = m_lctx.mk_local_decl(ngen(), next_name(), type, e);
4566         m_fvars.push_back(fvar);
4567         return fvar;
4568     }
4569
4570     /* Return `let _x := e in _x` */
4571     expr mk_trivial_let(expr const &e) {
4572         expr type = infer_type(e);
4573         return ::lean::mk_let("_x", type, e, mk_bvar(0));
4574     }
4575
4576     /* Create minor premise in LCNF.
4577        The minor premise is of the form `fun xs, e`.
4578        However, if `e` is a lambda, we create `fun xs, let _x := e in _x`.
4579        Thus, we don't "mix" `xs` variables with
4580        the variables of the `new_minor` lambda */
4581     expr mk_minor_lambda(buffer<expr> const &xs, expr e) {
4582         if (is_lambda(e)) {
4583             /* We don't want to "mix" `xs` variables with
4584                the variables of the `new_minor` lambda */
4585             e = mk_trivial_let(e);
4586         }
4587         return m_lctx.mk_lambda(xs, e);
4588     }
4589
4590     /* See `mk_minor_lambda`. We want to preserve the arity of join-points. */
4591     expr mk_join_point_lambda(buffer<expr> const &xs, expr e) {
4592         return mk_minor_lambda(xs, e);
4593     }
4594
4595     expr get_lambda_body(expr e, buffer<expr> &xs) {
4596         while (is_lambda(e)) {
4597             expr d = instantiate_rev(binding_domain(e), xs.size(), xs.data());
4598             expr x = m_lctx.mk_local_decl(ngen(), binding_name(e), d,
4599                                           binding_info(e));
4600             xs.push_back(x);
4601             e = binding_body(e);
4602         }
4603         return instantiate_rev(e, xs.size(), xs.data());
4604     }
4605
4606     expr get_minor_body(expr e, buffer<expr> &xs) {
4607         unsigned i = 0;
4608         while (is_lambda(e)) {
4609             expr d = instantiate_rev(binding_domain(e), xs.size(), xs.data());
4610             expr x = m_lctx.mk_local_decl(ngen(), binding_name(e), d,
4611                                           binding_info(e));
4612             xs.push_back(x);
4613             i++;
4614             e = binding_body(e);
4615         }
4616         return instantiate_rev(e, xs.size(), xs.data());
4617     }
4618
4619     /* Move let-decl `fvar` to the minor premise at position `minor_idx` of
```

```
4620      * cases_on-application `c`. */
4621     expr move_let_to_minor(expr const &c, unsigned minor_idx,
4622                            expr const &fvar) {
4623         lean_assert(is_cases_on_app(env(), c));
4624         buffer<expr> args;
4625         expr const &c_fn = get_app_args(c, args);
4626         expr minor = args[minor_idx];
4627         buffer<expr> xs;
4628         minor = get_lambda_body(minor, xs);
4629         if (minor == fvar) {
4630             /* `let x := v in x` ==> `v` */
4631             minor = *m_lctx.get_local_decl(fvar).get_value();
4632         } else {
4633             xs.push_back(fvar);
4634         }
4635         args[minor_idx] = mk_minor_lambda(xs, minor);
4636         return mk_app(c_fn, args);
4637     }
4638
4639     /* Collect information for deciding whether `float_cases_on` is useful or
4640        not, and control code blowup. */
4641     struct cases_info_result {
4642         /* The number of branches takes into account join-points too. That is,
4643            it is not just the number of minor premises. */
4644         unsigned m_num_branches{0};
4645         /* The number of branches that return a constructor application. */
4646         unsigned m_num_cnstr_results{0};
4647         name_hash_set m_visited_jps;
4648     };
4649
4650     void collect_cases_info(expr e, cases_info_result &result) {
4651         while (true) {
4652             if (is_lambda(e))
4653                 e = binding_body(e);
4654             else if (is_let(e))
4655                 e = let_body(e);
4656             else
4657                 break;
4658         }
4659         if (is_constructor_app(env(), e)) {
4660             result.m_num_branches++;
4661             result.m_num_cnstr_results++;
4662         } else if (is_cases_on_app(env(), e)) {
4663             buffer<expr> args;
4664             expr const &fn = get_app_args(e, args);
4665             unsigned begin_minors;
4666             unsigned end_minors;
4667             std::tie(begin_minors, end_minors) = get_cases_on_minors_range(
4668                 env(), const_name(fn), m_before_erasure);
4669             for (unsigned i = begin_minors; i < end_minors; i++) {
4670                 collect_cases_info(args[i], result);
4671             }
4672         } else if (is_join_point_app(e)) {
4673             expr const &fn = get_app_fn(e);
4674             lean_assert(is_fvar(fn));
4675             if (result.m_visited_jps.find(fvar_name(fn)) !=
4676                 result.m_visited_jps.end())
4677                 return;
4678             result.m_visited_jps.insert(fvar_name(fn));
4679             local_decl decl = m_lctx.get_local_decl(fn);
4680             collect_cases_info(*decl.get_value(), result);
4681         } else {
4682             result.m_num_branches++;
4683         }
4684     }
4685
4686     /* The `float_cases_on` transformation may produce code duplication.
4687        The term `e` is "copied" in each branch of the the `cases_on` expression
4688        `c`. This method creates one (or more) join-point(s) for `e` (if needed).
4689        Return `none` if the code size increase is above the threshold.
```

```
4690        Remark: it may produce type incorrect terms. */
4691     expr mk_join_point_float_cases_on(expr const &fvar, expr const &e,
4692                                       expr const &c) {
4693         lean_assert(is_cases_on_app(env(), c));
4694         unsigned e_size = get_lcnf_size(env(), e);
4695         if (e_size == 1) {
4696             return e;
4697         }
4698         cases_info_result c_info;
4699         collect_cases_info(c, c_info);
4700         unsigned code_increase = e_size * (c_info.m_num_branches - 1);
4701         if (code_increase <= m_cfg.m_float_cases_threshold) {
4702             return e;
4703         }
4704         local_decl fvar_decl = m_lctx.get_local_decl(fvar);
4705         if (is_cases_on_app(env(), e)) {
4706             buffer<expr> args;
4707             expr const &fn = get_app_args(e, args);
4708             inductive_val e_I_val = get_cases_on_inductive_val(env(), fn);
4709             /* We can control the code blowup by creating join points for each
4710                branch. In the worst case, each branch becomes a join point jump,
4711                and the
4712                "compressed size" is equal to the number of branches + 1 for the
4713                cases_on application. */
4714             unsigned e_compressed_size = e_I_val.get_ncnstrs() + 1;
4715             /* We can ignore the cost of branches that return constructors since
4716                they will in the worst case become join point jumps. */
4717             unsigned new_code_increase =
4718                 e_compressed_size *
4719                 (c_info.m_num_branches - c_info.m_num_cnstr_results);
4720             if (new_code_increase <= m_cfg.m_float_cases_threshold) {
4721                 unsigned branch_threshold =
4722                     m_cfg.m_float_cases_threshold / (c_info.m_num_branches - 1);
4723                 unsigned begin_minors;
4724                 unsigned end_minors;
4725                 std::tie(begin_minors, end_minors) = get_cases_on_minors_range(
4726                     env(), const_name(fn), m_before_erasure);
4727                 for (unsigned minor_idx = begin_minors; minor_idx < end_minors;
4728                      minor_idx++) {
4729                     expr minor = args[minor_idx];
4730                     if (get_lcnf_size(env(), minor) > branch_threshold) {
4731                         buffer<bool>
4732                             used_zs; /* used_zs[i] iff `minor` uses `zs[i]` */
4733                         bool used_fvar =
4734                             false; /* true iff `minor` uses `fvar` */
4735                         bool used_unit = false; /* true if we needed to add
4736                                                    `unit ->` to joint point */
4737                         expr jp_val;
4738                         /* Create join-point value: `jp-val` */
4739                         {
4740                             buffer<expr> zs;
4741                             minor = get_lambda_body(minor, zs);
4742                             mark_used_fvars(minor, zs, used_zs);
4743                             lean_assert(zs.size() == used_zs.size());
4744                             used_fvar = false;
4745                             jp_val = minor;
4746                             buffer<expr> jp_args;
4747                             if (has_fvar(minor, fvar)) {
4748                                 /* `fvar` is a let-decl variable, we need to
4749                                    convert into a lambda variable. Remark: we
4750                                    need to use `replace_fvar_with` because
4751                                    replacing the let-decl variable `fvar` with
4752                                    the lambda variable `new_fvar` may produce a
4753                                    type incorrect term. */
4754                                 used_fvar = true;
4755                                 expr new_fvar = m_lctx.mk_local_decl(
4756                                     ngen(), fvar_decl.get_user_name(),
4757                                     fvar_decl.get_type());
4758                                 jp_args.push_back(new_fvar);
4759                                 jp_val = replace_fvar(jp_val, fvar, new_fvar);
```

```
4760                            }
4761                            for (unsigned i = 0; i < used_zs.size(); i++) {
4762                                if (used_zs[i]) jp_args.push_back(zs[i]);
4763                            }
4764                            if (jp_args.empty()) {
4765                                jp_args.push_back(m_lctx.mk_local_decl(
4766                                    ngen(), "_", mk_unit()));
4767                                used_unit = true;
4768                            }
4769                            jp_val = mk_join_point_lambda(jp_args, jp_val);
4770                        }
4771                        /* Create new jp */
4772                        expr jp_type = cheap_beta_reduce(infer_type(jp_val));
4773                        mark_simplified(jp_val);
4774                        expr jp_var = m_lctx.mk_local_decl(
4775                            ngen(), next_jp_name(), jp_type, jp_val);
4776                        register_new_jp(jp_var);
4777                        /* Replace minor with new jp */
4778                        {
4779                            buffer<expr> zs;
4780                            minor = args[minor_idx];
4781                            minor = get_lambda_body(minor, zs);
4782                            lean_assert(zs.size() == used_zs.size());
4783                            expr new_minor = jp_var;
4784                            if (used_unit)
4785                                new_minor = mk_app(new_minor, mk_unit_mk());
4786                            if (used_fvar) new_minor = mk_app(new_minor, fvar);
4787                            for (unsigned i = 0; i < used_zs.size(); i++) {
4788                                if (used_zs[i])
4789                                    new_minor = mk_app(new_minor, zs[i]);
4790                            }
4791                            new_minor = mk_minor_lambda(zs, new_minor);
4792                            args[minor_idx] = new_minor;
4793                        }
4794                    }
4795                }
4796                lean_trace(name({"compiler", "simp_float_cases"}),
4797                           tout() << "mk_join " << fvar << "\n"
4798                                  << c << "\n---\n"
4799                                  << e << "\n======>\n"
4800                                  << mk_app(fn, args) << "\n";);
4801                return mk_app(fn, args);
4802            }
4803        }
4804        /* Create simple join point */
4805        expr jp_val = e;
4806        if (is_lambda(e)) jp_val = mk_trivial_let(jp_val);
4807        jp_val =
4808            ::lean::mk_lambda(fvar_decl.get_user_name(), fvar_decl.get_type(),
4809                              abstract(jp_val, fvar));
4810        expr jp_type = cheap_beta_reduce(infer_type(jp_val));
4811        mark_simplified(jp_val);
4812        expr jp_var =
4813            m_lctx.mk_local_decl(ngen(), next_jp_name(), jp_type, jp_val);
4814        register_new_jp(jp_var);
4815        return mk_app(jp_var, fvar);
4816    }
4817
4818    /* Given `e[x]`, create a let-decl `y := v`, and return `e[y]`
4819       Note that, this transformation may produce type incorrect terms.
4820
4821       Remove: if `v` is an atom, we do not create `y`. */
4822    expr apply_at(expr const &x, expr const &e, expr const &v) {
4823        if (is_lcnf_atom(v)) {
4824            expr e_v = replace_fvar(e, x, v);
4825            return visit(e_v, false);
4826        } else {
4827            local_decl x_decl = m_lctx.get_local_decl(x);
4828            expr y = m_lctx.mk_local_decl(ngen(), x_decl.get_user_name(),
4829                                          x_decl.get_type(), v);
```

```
4830            expr e_y = replace_fvar(e, x, y);
4831            m_fvars.push_back(y);
4832            return visit(e_y, false);
4833        }
4834    }
4835
4836    expr_pair mk_jp_cache_key(expr const &x, expr const &e, expr const &jp) {
4837        expr x_type = m_lctx.get_local_decl(x).get_type();
4838        expr abst_e = ::lean::mk_lambda("_x", x_type, abstract(e, x));
4839        return mk_pair(abst_e, jp);
4840    }
4841
4842    /*
4843      Given `e[x]`
4844      ```
4845      let jp := fun z, let .... in e'
4846      ```
4847      ==>
4848      ```
4849      let jp' := fun z, let ... y := e' in e[y]
4850      ```
4851      If `e'` is a `cases_on` application, we use `float_cases_on_core`. That
4852      is,
4853      ```
4854      let jp := fun z, let ... in
4855                cases_on m
4856                 (fun y_1, let ... in e_1)
4857                 ...
4858                 (fun y_n, let ... in e_n)
4859      ```
4860      ==>
4861      ```
4862      let jp := fun z, let ... in
4863                cases_on m
4864                 (fun y_1, let ... y := e_1 in e[y])
4865                 ...
4866                 (fun y_n, let ... y := e_n in e[y])
4867      ```
4868
4869      Remark: this method may produce type incorrect terms because of dependent
4870      types. */
4871    expr mk_new_join_point(expr const &x, expr const &e, expr const &jp) {
4872        expr_pair key = mk_jp_cache_key(x, e, jp);
4873        auto it = m_jp_cache.find(key);
4874        if (it != m_jp_cache.end()) return it->second;
4875        local_decl jp_decl = m_lctx.get_local_decl(jp);
4876        lean_assert(is_join_point_name(jp_decl.get_user_name()));
4877        expr jp_val = *jp_decl.get_value();
4878        buffer<expr> zs;
4879        unsigned saved_fvars_size = m_fvars.size();
4880        jp_val = visit(get_lambda_body(jp_val, zs), false);
4881        expr e_y;
4882        if (is_join_point_app(jp_val)) {
4883            buffer<expr> jp2_args;
4884            expr const &jp2 = get_app_args(jp_val, jp2_args);
4885            expr new_jp2 = mk_new_join_point(x, e, jp2);
4886            e_y = mk_app(new_jp2, jp2_args);
4887        } else if (is_cases_on_app(env(), jp_val)) {
4888            e_y = float_cases_on_core(x, e, jp_val);
4889        } else {
4890            e_y = apply_at(x, e, jp_val);
4891        }
4892        expr new_jp_val = e_y;
4893        new_jp_val = mk_let(zs, saved_fvars_size, new_jp_val, false);
4894        new_jp_val = mk_join_point_lambda(zs, new_jp_val);
4895        mark_simplified(new_jp_val);
4896        expr new_jp_type = cheap_beta_reduce(infer_type(new_jp_val));
4897        expr new_jp_var = m_lctx.mk_local_decl(ngen(), next_jp_name(),
4898                                               new_jp_type, new_jp_val);
4899        register_new_jp(new_jp_var);
```

```
4900         m_jp_cache.insert(mk_pair(key, new_jp_var));
4901         return new_jp_var;
4902     }
4903
4904     /* Add entry `x := cidx fields` to m_expr2ctor */
4905     void update_expr2ctor(expr const &x, expr const &c_fn,
4906                           buffer<expr> const &c_args, unsigned cidx,
4907                           buffer<expr> const &fields) {
4908         inductive_val I_val = get_cases_on_inductive_val(env(), c_fn);
4909         name ctor_name = get_ith(I_val.get_cnstrs(), cidx);
4910         levels ctor_lvls;
4911         buffer<expr> ctor_args;
4912         if (m_before_erasure) {
4913             ctor_lvls = tail(const_levels(c_fn));
4914             ctor_args.append(I_val.get_nparams(), c_args.data());
4915         } else {
4916             for (unsigned i = 0; i < I_val.get_nparams(); i++)
4917                 ctor_args.push_back(mk_enf_neutral());
4918         }
4919         ctor_args.append(fields);
4920         expr ctor = mk_app(mk_constant(ctor_name, ctor_lvls), ctor_args);
4921         m_expr2ctor.insert(x, ctor);
4922     }
4923
4924     /* Given `e[x]`
4925       ```
4926       cases_on m
4927               (fun zs, let ... in e_1)
4928               ...
4929               (fun zs, let ... in e_n)
4930       ```
4931       ==>
4932       ```
4933       cases_on m
4934         (fun zs, let ... y := e_1 in e[y])
4935         ...
4936         (fun y_n, let ... y := e_n in e[y])
4937       ``` */
4938     expr float_cases_on_core(expr const &x, expr const &e, expr const &c) {
4939         lean_assert(is_cases_on_app(env(), c));
4940         local_decl x_decl = m_lctx.get_local_decl(x);
4941         buffer<expr> c_args;
4942         expr c_fn = get_app_args(c, c_args);
4943         inductive_val I_val = get_cases_on_inductive_val(env(), c_fn);
4944         unsigned major_idx;
4945         /* Update motive and get major_idx */
4946         if (m_before_erasure) {
4947             unsigned motive_idx = I_val.get_nparams();
4948             unsigned first_index = motive_idx + 1;
4949             unsigned nindices = I_val.get_nindices();
4950             major_idx = first_index + nindices;
4951             buffer<expr> zs;
4952             expr result_type = whnf_infer_type(e);
4953             expr motive = c_args[motive_idx];
4954             expr motive_type = whnf_infer_type(motive);
4955             for (unsigned i = 0; i < nindices + 1; i++) {
4956                 lean_assert(is_pi(motive_type));
4957                 expr z = m_lctx.mk_local_decl(ngen(), binding_name(motive_type),
4958                                               binding_domain(motive_type),
4959                                               binding_info(motive_type));
4960                 zs.push_back(z);
4961                 motive_type = whnf(instantiate(binding_body(motive_type), z));
4962             }
4963             level result_lvl = sort_level(tc().ensure_type(result_type));
4964             if (has_fvar(result_type, x)) {
4965                 /* `x` will be deleted after the float_cases_on transformation.
4966                    So, if the result type depends on it, we must replace it with
4967                    its value. */
4968                 result_type = replace_fvar(result_type, x, *x_decl.get_value());
4969             }
```

```
4970            expr new_motive = m_lctx.mk_lambda(zs, result_type);
4971            c_args[motive_idx] = new_motive;
4972            /* We need to update the resultant universe. */
4973            levels new_cases_lvls =
4974                levels(result_lvl, tail(const_levels(c_fn)));
4975            c_fn = update_constant(c_fn, new_cases_lvls);
4976        } else {
4977            /* After erasure, we keep only major and minor premises. */
4978            major_idx = 0;
4979        }
4980        /* Update minor premises */
4981        expr const &major = c_args[major_idx];
4982        unsigned first_minor_idx = major_idx + 1;
4983        unsigned nminors = I_val.get_ncnstrs();
4984        for (unsigned i = 0; i < nminors; i++) {
4985            unsigned minor_idx = first_minor_idx + i;
4986            expr minor = c_args[minor_idx];
4987            buffer<expr> zs;
4988            unsigned saved_fvars_size = m_fvars.size();
4989            expr minor_val = get_minor_body(minor, zs);
4990            {
4991                flet<expr2ctor> save_expr2ctor(m_expr2ctor, m_expr2ctor);
4992                update_expr2ctor(major, c_fn, c_args, i, zs);
4993                minor_val = visit(minor_val, false);
4994            }
4995            expr new_minor;
4996            if (is_join_point_app(minor_val)) {
4997                buffer<expr> jp_args;
4998                expr const &jp = get_app_args(minor_val, jp_args);
4999                expr new_jp = mk_new_join_point(x, e, jp);
5000                new_minor = visit(mk_app(new_jp, jp_args), false);
5001            } else {
5002                new_minor = apply_at(x, e, minor_val);
5003            }
5004            new_minor = mk_let(zs, saved_fvars_size, new_minor, false);
5005            new_minor = mk_minor_lambda(zs, new_minor);
5006            c_args[minor_idx] = new_minor;
5007        }
5008        lean_trace(name({"compiler", "simp_float_cases"}),
5009                   tout() << "float_cases_on [" << get_lcnf_size(env(), e)
5010                          << "]\n"
5011                          << c << "\n----\n"
5012                          << e << "\n=====>\n"
5013                          << mk_app(c_fn, c_args) << "\n";);
5014        return mk_app(c_fn, c_args);
5015    }
5016
5017    /* Float cases transformation (see: `float_cases_on_core`).
5018       This version may create join points if `e` is big, or "good" join-points
5019       could not be created. */
5020    expr float_cases_on(expr const &x, expr const &e, expr const &c) {
5021        expr new_e = mk_join_point_float_cases_on(x, e, c);
5022        return float_cases_on_core(x, new_e, c);
5023    }
5024
5025    /* Given the buffer `entries`: `[(x_1, w_1), ..., (x_n, w_n)]`, and `e`.
5026       Create the let-expression
5027       ```
5028       let x_n := w_n
5029           ...
5030           x_1 := w_1
5031       in e
5032       ```
5033       The values `w_i` are the "simplified values" for the let-declaration
5034       `x_i`. */
5035    expr mk_let_core(buffer<pair<expr, expr>> const &entries, expr e) {
5036        buffer<expr> fvars;
5037        buffer<name> user_names;
5038        buffer<expr> types;
5039        buffer<expr> vals;
```

```
5040         unsigned i = entries.size();
5041         while (i > 0) {
5042             --i;
5043             expr const &fvar = entries[i].first;
5044             fvars.push_back(fvar);
5045             expr const &val = entries[i].second;
5046             vals.push_back(val);
5047             local_decl fvar_decl = m_lctx.get_local_decl(fvar);
5048             user_names.push_back(fvar_decl.get_user_name());
5049             types.push_back(fvar_decl.get_type());
5050         }
5051         e = abstract(e, fvars.size(), fvars.data());
5052         i = fvars.size();
5053         while (i > 0) {
5054             --i;
5055             expr new_value = abstract(vals[i], i, fvars.data());
5056             expr new_type = abstract(types[i], i, fvars.data());
5057             e = ::lean::mk_let(user_names[i], new_type, new_value, e);
5058         }
5059         return e;
5060     }
5061
5062     /* Split `entries` into two groups: `entries_dep_x` and `entries_ndep_x`.
5063        The first group contains the entries that depend on `x` and the second
5064        the ones that doesn't. This auxiliary method is used to float cases_on
5065        over expressions.
5066
5067        `entries` is of the form `[(x_1, w_1), ..., (x_n, w_n)]`, where `x_i`s
5068        are let-decl free variables, and `w_i`s their new values. We use
5069        `entries` and an expression `e` to create a `let` expression:
5070        ```
5071        let x_n := w_n
5072            ...
5073            x_1 := w_1
5074        in e
5075        ``` */
5076     void split_entries(buffer<pair<expr, expr>> const &entries, expr const &x,
5077                        buffer<pair<expr, expr>> &entries_dep_x,
5078                        buffer<pair<expr, expr>> &entries_ndep_x) {
5079         if (entries.empty()) return;
5080         name_hash_set deps;
5081         deps.insert(fvar_name(x));
5082         /* Recall that `entries` are in reverse order. That is, pos 0 is the
5083          * inner most variable. */
5084         unsigned i = entries.size();
5085         while (i > 0) {
5086             --i;
5087             expr const &fvar = entries[i].first;
5088             expr fvar_type = m_lctx.get_type(fvar);
5089             expr fvar_new_val = entries[i].second;
5090             if (depends_on(fvar_type, deps) || depends_on(fvar_new_val, deps)) {
5091                 deps.insert(fvar_name(fvar));
5092                 entries_dep_x.push_back(entries[i]);
5093             } else {
5094                 entries_ndep_x.push_back(entries[i]);
5095             }
5096         }
5097         std::reverse(entries_dep_x.begin(), entries_dep_x.end());
5098         std::reverse(entries_ndep_x.begin(), entries_ndep_x.end());
5099     }
5100
5101     bool push_dep_jps(expr const &fvar) {
5102         lean_assert(is_fvar(fvar));
5103         auto it = m_fvar2jps.find(fvar);
5104         if (it == m_fvar2jps.end()) return false;
5105         buffer<expr> tmp;
5106         to_buffer(it->second, tmp);
5107         m_fvar2jps.erase(fvar);
5108         std::reverse(tmp.begin(), tmp.end());
5109         m_fvars.append(tmp);
```

```
5110        return true;
5111    }
5112
5113    bool push_dep_jps(buffer<expr> const &zs, bool top) {
5114        buffer<expr> tmp;
5115        if (top) {
5116            to_buffer(m_closed_jps, tmp);
5117            m_closed_jps = exprs();
5118        }
5119        for (expr const &z : zs) {
5120            auto it = m_fvar2jps.find(z);
5121            if (it != m_fvar2jps.end()) {
5122                to_buffer(it->second, tmp);
5123                m_fvar2jps.erase(z);
5124            }
5125        }
5126        if (tmp.empty()) return false;
5127        sort_fvars(m_lctx, tmp);
5128        m_fvars.append(tmp);
5129        return true;
5130    }
5131
5132    void sort_entries(buffer<expr_pair> &entries) {
5133        std::sort(entries.begin(), entries.end(),
5134                  [&](expr_pair const &p1, expr_pair const &p2) {
5135                      /* We use `>` because entries in `entries` are in reverse
5136                       * dependency order */
5137                      return m_lctx.get_local_decl(p1.first).get_idx() >
5138                             m_lctx.get_local_decl(p2.first).get_idx();
5139                  });
5140    }
5141
5142    /* Copy `src_entries` and the new joint points that depend on them to
5143       `entries`, and update `entries_fvars`. This method is used after we
5144       perform a `float_cases_on`. */
5145    void move_to_entries(buffer<expr_pair> const &src_entries,
5146                         buffer<expr_pair> &entries,
5147                         name_hash_set &entries_fvars) {
5148        buffer<expr_pair> todo;
5149        for (unsigned i = 0; i < src_entries.size(); i++) {
5150            expr_pair const &entry = src_entries[i];
5151            /* New join points may have been attached to `ndep_entry` */
5152            todo.push_back(entry);
5153            while (!todo.empty()) {
5154                expr_pair const &curr = todo.back();
5155                auto it = m_fvar2jps.find(curr.first);
5156                if (it != m_fvar2jps.end()) {
5157                    buffer<expr> tmp;
5158                    to_buffer(it->second, tmp);
5159                    for (expr const &jp : tmp) {
5160                        /* Recall that new join points have already been
5161                           simplified. So, it is ok to move them to `entries`.
5162                         */
5163                        todo.emplace_back(
5164                            jp, *m_lctx.get_local_decl(jp).get_value());
5165                    }
5166                    m_fvar2jps.erase(curr.first);
5167                } else {
5168                    entries.push_back(curr);
5169                    collect_used(curr.second, entries_fvars);
5170                    todo.pop_back();
5171                }
5172            }
5173        }
5174        /* The following sorting operation is necessary because of non trivial
5175           dependencies between entries. For example, consider the following
5176           scenario. When starting a `float_cases_on` operation, we determine
5177           that the already processed entries `[_j_1._join, _x_1]` do not depend
5178           on the operation. Moreover, `_j_1._join` is a new join-point that
5179           depends on `_x_1`. Recall that entries are in reverse dependecy
```

```
5180            order, and this is why `_j_1._join` occurs before `_x_1`. Then,
5181            during the actual execution of the `float_cases_on` operation, we
5182            create a new joint point `_j_2._join` that depends on `_j_1._join`,
5183            and is consequently attached to `_x_1`, that is, `m_fvar2jps[_x_1]`
5184            contains `_j_2._join`. After executing this procedure, `entries` will
5185            contain `[_j_1._join, _j_2._join, _x_1]` which is incorrect since
5186            `_j_2._join` depends on `_j_1._join`. */
5187         sort_entries(entries);
5188     }
5189
5190     /* Given a casesOn application `c`, return `some idx` iff `c` has more than
5191        one branch, `fvar` only occurs in the argument `idx`, this argument is a
5192        minor premise.
5193
5194        Recall this method is used to implement the float `let` inwards
5195        transformation. Thus, it doesn't really help to move `let` inwards if
5196        there is only one branch.
5197
5198        Moreover, it may negatively impact performance because we use `casesOn`
5199        applications to guide the insertion of reset/reuse IR instructions.
5200
5201        Here is a problematic example:
5202        ```
5203        let p := Array.index a i in                -- Get pair `p` at `a[i]`
5204        let a := Array.update a i (default _) in   -- "Reset" `a[i]` to make sure
5205        `p` is now the owner casesOn p (fun fst snd, Array.update a i (fst+1,
5206        snd))
5207        ```
5208        Before this commit the compiler would move
5209        ```
5210        a := Array.update a i (default _)
5211        ```
5212        into the `casesOn` branch, and we would get
5213        ```
5214        let p := Array.index a i in                -- Get pair `p` at `a[i]`
5215        casesOn p (fun fst snd,
5216          let a := Array.update a i (default _) in -- "Reset" `a[i]` to make sure
5217        `p` is now the owner Array.update a i (fst+1, snd))
5218        ```
5219        Then, we would get
5220        ```
5221        let p := Array.index a i in                -- Get pair `p` at `a[i]`
5222        casesOn p (fun fst snd,
5223          let p := reset p in
5224          let a := Array.update a i (default _) in -- "Reset" `a[i]` to make sure
5225        `p` is now the owner let p := reuse p (fst+1, snd) in Array.update a i p)
5226        ```
5227        But, this `reset p` will always fail since the `Array` still contains a
5228        reference to `p` when we execute `reset p`.
5229     */
5230     optional<unsigned> used_in_one_minor(expr const &c, expr const &fvar) {
5231         lean_assert(is_cases_on_app(env(), c));
5232         lean_assert(is_fvar(fvar));
5233         buffer<expr> args;
5234         expr const &c_fn = get_app_args(c, args);
5235         unsigned minors_begin;
5236         unsigned minors_end;
5237         std::tie(minors_begin, minors_end) = get_cases_on_minors_range(
5238             env(), const_name(c_fn), m_before_erasure);
5239         if (minors_end <= minors_begin + 1) {
5240             /* casesOn has only one branch */
5241             return optional<unsigned>();
5242         }
5243         unsigned i = 0;
5244         for (; i < minors_begin; i++) {
5245             if (has_fvar(args[i], fvar)) {
5246                 /* Free variable occurs in a term that is a not a minor premise.
5247                  */
5248                 return optional<unsigned>();
5249             }
```

```
5250         }
5251         lean_assert(i == minors_begin);
5252 /* The following #pragma is to disable a bogus g++ 4.9 warning at
5253  * `optional<unsigned> r` */
5254 #if defined(__GNUC__) && !defined(__CLANG__)
5255 #pragma GCC diagnostic ignored "-Wmaybe-uninitialized"
5256 #endif
5257         optional<unsigned> r;
5258         for (; i < minors_end; i++) {
5259             expr minor = args[i];
5260             while (is_lambda(minor)) {
5261                 if (has_fvar(binding_domain(minor), fvar)) {
5262                     /* Free variable occurs in the type of a field */
5263                     return optional<unsigned>();
5264                 }
5265                 minor = binding_body(minor);
5266             }
5267             if (has_fvar(minor, fvar)) {
5268                 if (r) {
5269                     /* Free variable occur in more than one minor premise. */
5270                     return optional<unsigned>();
5271                 }
5272                 r = i;
5273             }
5274         }
5275         return r;
5276     }
5277
5278     /*
5279       Given `x := val`, the entries `y_1 := w_1; ...; y_n := w_n`, and the set
5280       `S` of all free variables in `entries`. Return true if we may move `x :=
5281       val` after these entries.
5282
5283       This method is used to implement the float `let` inwards transformation.
5284     */
5285     bool may_move_after(expr const &x, expr const & /* val */,
5286                         buffer<expr_pair> const &entries,
5287                         name_hash_set const &S) {
5288         lean_assert(is_fvar(x));
5289         if (S.find(fvar_name(x)) != S.end()) {
5290             /* If `x` is used in the entries `y_1 := w_1; ...; y_n := w_n`,
5291                then we must *not* move `x` after them since it would produce
5292                an ill-formed expression. */
5293             return false;
5294         }
5295         /* The condition above is sufficient to make sure the resulting
5296            expression is well-formed. However, moving `x := val` after `entries`
5297            may affect perform by preventing destructive updates from happening
5298            and memory from being reused. Consider the following example
5299            ```
5300            let x := z.1 in
5301            let y := f z in
5302            C
5303            ```
5304            If we move `x := z.1` after `y := f z` obtaining the expression:
5305            ```
5306            let y := f z in
5307            let x := z.1 in
5308            C
5309            ```
5310            Then, `RC(z)` will be greater than 1 when we invoke `f z` because we
5311            would need to include an `inc z` instruction before `y := f z`. The
5312            `inc z` is needed because `z` would still be alive after `f z`.
5313
5314            In the example above, `val` contains a variable (`z`) used in
5315            `entries`. However, this test is not sufficient. Here is a more
5316            intricate example:
5317            ```
5318            let w := z.1 in
5319            let x := Array.size w in
```

```
5320           let y := f z in
5321           C
5322           ```
5323           If we move `x := Array.size w` after `y := f z`, we get
5324           ```
5325           let w := z.1 in
5326           let y := f z in
5327           let x := Array.size w in
5328           C
5329           ```
5330           `f z` and `Array.size w` do not share any free variable, but `w` is
5331           an reference to a field of `z`. In the example above, `w` is an
5332           array, and `f z` will not be able to update the array nested there if
5333           we have `let x := Array.size w` after it.
5334
5335           The example above suggests that a sufficient condition for preventing
5336           this issue is:
5337           - Any memory cell reachable from `val` is not reachable from
5338           `entries`.
5339
5340           A simpler sufficient condition for preventing the issue is:
5341           - `entries` code does not perform destructive updates or tries to
5342           reuse memory cells. Here we use an even simpler check: `entries`
5343           contains only projection operations.
5344       */
5345       for (expr_pair const &p : entries) {
5346           expr const &w = p.second;
5347           if (!is_proj(w)) return false;
5348       }
5349       return true;
5350   }
5351
5352   /* Create a let-expression with body `e`, and
5353      all "used" let-declarations `m_fvars[i]` for `i in [saved_fvars_size,
5354      m_fvars.size)`. We also include all join points that depends on these
5355      free variables, nad join points that depends on `zs`. The buffer `zs`
5356      (when non empty) contains the free variables for a lambda expression that
5357      will be created around the let-expression.
5358
5359      BTW, we also visit the lambda expressions in used let-declarations of the
5360      form `x : t := fun ...`
5361
5362
5363      Note that, we don't visit them when we have visit let-expressions. */
5364   expr mk_let(buffer<expr> const &zs, unsigned saved_fvars_size, expr e,
5365               bool top) {
5366       if (saved_fvars_size == m_fvars.size()) {
5367           if (!push_dep_jps(zs, top)) return e;
5368       }
5369       /* `entries` contains pairs (let-decl fvar, new value) for building the
5370          resultant let-declaration. We simplify the value of some
5371          let-declarations in this method, but we don't want to create a new
5372          temporary declaration just for this. */
5373       buffer<expr_pair> entries;
5374       name_hash_set e_fvars; /* Set of free variables names used in `e` */
5375       name_hash_set
5376           entries_fvars; /* Set of free variable names used in `entries` */
5377       collect_used(e, e_fvars);
5378       bool e_is_cases = is_cases_on_app(env(), e);
5379       /*
5380         Recall that all free variables in `m_fvars` are let-declarations.
5381         In the following loop, we have the following "order" for the
5382         let-declarations:
5383         ```
5384            m_fvars[saved_fvars_size]
5385            ...
5386            m_fvars[m_fvars.size() - 1]
5387
5388            entries[entries.size() - 1]
5389            ...
```

```
5390             entries[0]
5391           ```
5392           The "body" of the let-declaration is `e`.
5393           The mapping `m_fvar2jps` maps a free variable `x to join points that
5394           must be inserted after `x`.
5395         */
5396         while (true) {
5397             if (m_fvars.size() == saved_fvars_size) {
5398                 if (!push_dep_jps(zs, top)) break;
5399             }
5400             lean_assert(m_fvars.size() > saved_fvars_size);
5401             expr x = m_fvars.back();
5402             if (push_dep_jps(x)) {
5403                 /* We must process the join points that depend on `x` before we
5404                  * process `x`. */
5405                 continue;
5406             }
5407             m_fvars.pop_back();
5408             bool used_in_e = (e_fvars.find(fvar_name(x)) != e_fvars.end());
5409             bool used_in_entries =
5410                 (entries_fvars.find(fvar_name(x)) != entries_fvars.end());
5411             if (!used_in_e && !used_in_entries) {
5412                 /* Skip unused variables */
5413                 continue;
5414             }
5415             local_decl x_decl = m_lctx.get_local_decl(x);
5416             expr type = x_decl.get_type();
5417             expr val = *x_decl.get_value();
5418             bool is_jp = false;
5419             bool modified_val = false;
5420             if (is_lambda(val)) {
5421                 /* We don't simplify lambdas when we visit `let`-expressions. */
5422                 DEBUG_CODE(unsigned saved_fvars_size = m_fvars.size(););
5423                 is_jp = is_join_point_name(x_decl.get_user_name());
5424                 val = visit_lambda(val, is_jp, false);
5425                 modified_val = true;
5426                 lean_assert(m_fvars.size() == saved_fvars_size);
5427             }
5428
5429             if (is_lc_unreachable_app(val)) {
5430                 /* `let x := lc_unreachable in e` => `lc_unreachable` */
5431                 e = val;
5432                 e_is_cases = false;
5433                 e_fvars.clear();
5434                 entries_fvars.clear();
5435                 collect_used(e, e_fvars);
5436                 entries.clear();
5437                 continue;
5438             }
5439
5440             if (entries.empty() && e == x) {
5441                 /* `let x := v in x` ==> `v` */
5442                 e = val;
5443                 collect_used(val, e_fvars);
5444                 e_is_cases = is_cases_on_app(env(), e);
5445                 continue;
5446             }
5447
5448             if (is_cases_on_app(env(), val)) {
5449                 /* We first create a let-declaration with all entries that
5450                    depends on the current `x` which is a cases_on application.
5451                  */
5452                 buffer<pair<expr, expr>> entries_dep_curr;
5453                 buffer<pair<expr, expr>> entries_ndep_curr;
5454                 split_entries(entries, x, entries_dep_curr, entries_ndep_curr);
5455                 expr new_e = mk_let_core(entries_dep_curr, e);
5456                 e = float_cases_on(x, new_e, val);
5457                 lean_assert(is_cases_on_app(env(), e));
5458                 e_is_cases = true;
5459                 /* Reset `e_fvars` and `entries_fvars`, we need to reconstruct
```

```
5460                 * them. */
5461                e_fvars.clear();
5462                entries_fvars.clear();
5463                collect_used(e, e_fvars);
5464                entries.clear();
5465                /* Copy `entries_ndep_curr` to `entries` */
5466                move_to_entries(entries_ndep_curr, entries, entries_fvars);
5467                continue;
5468            }
5469
5470            if (!is_jp && e_is_cases && used_in_e) {
5471                optional<unsigned> minor_idx = used_in_one_minor(e, x);
5472                if (minor_idx &&
5473                    may_move_after(x, val, entries, entries_fvars)) {
5474                    /* If x is only used in only one minor declaration,
5475                       and it passed the may_move_after test. */
5476                    if (modified_val) {
5477                        /* We need to create a new free variable since the new
5478                           simplified value `val` */
5479                        expr new_x = m_lctx.mk_local_decl(
5480                            ngen(), x_decl.get_user_name(), type, val);
5481                        e = replace_fvar(e, x, new_x);
5482                        x = new_x;
5483                    }
5484                    collect_used(type, e_fvars);
5485                    collect_used(val, e_fvars);
5486                    e = move_let_to_minor(e, *minor_idx, x);
5487                    continue;
5488                }
5489            }
5490
5491            collect_used(type, entries_fvars);
5492            collect_used(val, entries_fvars);
5493            entries.emplace_back(x, val);
5494        }
5495        return mk_let_core(entries, e);
5496    }
5497
5498    name mk_let_name(name const &n) {
5499        if (is_internal_name(n)) {
5500            if (is_join_point_name(n))
5501                return next_jp_name();
5502            else if (is_pseudo_do_join_point_name(n))
5503                return n;
5504            else
5505                return next_name();
5506        } else {
5507            return n;
5508        }
5509    }
5510
5511    expr visit_let(expr e) {
5512        buffer<expr> let_fvars;
5513        while (is_let(e)) {
5514            expr new_type = instantiate_rev(let_type(e), let_fvars.size(),
5515                                            let_fvars.data());
5516            expr new_val = visit(instantiate_rev(let_value(e), let_fvars.size(),
5517                                                 let_fvars.data()),
5518                                 true);
5519            if (!is_pseudo_do_join_point_name(let_name(e)) &&
5520                is_lcnf_atom(new_val)) {
5521                let_fvars.push_back(new_val);
5522            } else {
5523                name n = mk_let_name(let_name(e));
5524                expr new_fvar =
5525                    m_lctx.mk_local_decl(ngen(), n, new_type, new_val);
5526                let_fvars.push_back(new_fvar);
5527                m_fvars.push_back(new_fvar);
5528            }
5529            e = let_body(e);
```

```
5530         }
5531         return visit(instantiate_rev(e, let_fvars.size(), let_fvars.data()),
5532                      false);
5533     }
5534
5535     /* - `is_join_point_def` is true if the lambda is the value of a join point.
5536        - `root` is true if the lambda is the value of a definition. */
5537     expr visit_lambda(expr e, bool is_join_point_def, bool top) {
5538         lean_assert(is_lambda(e));
5539         lean_assert(!top || m_fvars.size() == 0);
5540         if (already_simplified(e)) return e;
5541         // Hack to avoid eta-expansion of implicit lambdas
5542         // Example: `fun {a} => ReaderT.pure`
5543         if (!is_join_point_def && !top) {
5544             expr new_e = eta_reduce(e);
5545             if (is_app(new_e) && !is_constructor_app(env(), new_e) &&
5546                 !is_proj(new_e) && !is_cases_on_app(env(), new_e) &&
5547                 !is_lc_unreachable_app(new_e))
5548                 return visit(new_e, true);
5549         }
5550         buffer<expr> binding_fvars;
5551         while (is_lambda(e)) {
5552             /* Types are ignored in compilation steps. So, we do not invoke
5553              * visit for d. */
5554             expr new_d = instantiate_rev(
5555                 binding_domain(e), binding_fvars.size(), binding_fvars.data());
5556             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e), new_d,
5557                                                  binding_info(e));
5558             binding_fvars.push_back(new_fvar);
5559             e = binding_body(e);
5560         }
5561         e = instantiate_rev(e, binding_fvars.size(), binding_fvars.data());
5562         /* When we simplify before erasure, we eta-expand all lambdas which are
5563          * not join points. */
5564         buffer<expr> eta_args;
5565         if (m_before_erasure && !is_join_point_def) {
5566             expr e_type = whnf_infer_type(e);
5567             while (is_pi(e_type)) {
5568                 expr arg = m_lctx.mk_local_decl(ngen(), binding_name(e_type),
5569                                                 binding_domain(e_type),
5570                                                 binding_info(e_type));
5571                 eta_args.push_back(arg);
5572                 e_type = whnf(instantiate(binding_body(e_type), arg));
5573             }
5574         }
5575         unsigned saved_fvars_size = m_fvars.size();
5576         expr new_body = visit(e, false);
5577         if (!eta_args.empty()) {
5578             if (is_join_point_app(new_body)) {
5579                 /* Remark: we cannot simply set
5580                    ```
5581                    new_body = mk_app(new_body, eta_args);
5582                    ```
5583                    when `new_body` is a join-point, because the result will not
5584                    be a valid LCNF term. We could expand the join-point, but it
5585                    this will create a copy. So, for now, we simply avoid
5586                    eta-expansion.
5587                 */
5588                 eta_args.clear();
5589             } else {
5590                 if (is_lcnf_atom(new_body)) {
5591                     new_body = mk_app(new_body, eta_args);
5592                 } else if (is_app(new_body) &&
5593                            !is_cases_on_app(env(), new_body)) {
5594                     new_body = mk_app(new_body, eta_args);
5595                 } else {
5596                     expr f = mk_let_decl(new_body);
5597                     new_body = mk_app(f, eta_args);
5598                 }
5599                 new_body = visit(new_body, false);
```

```
5600            }
5601            binding_fvars.append(eta_args);
5602        }
5603        new_body = mk_let(binding_fvars, saved_fvars_size, new_body, top);
5604        expr r;
5605        if (is_join_point_def) {
5606            lean_assert(eta_args.empty());
5607            r = mk_join_point_lambda(binding_fvars, new_body);
5608        } else {
5609            r = m_lctx.mk_lambda(binding_fvars, new_body);
5610        }
5611        mark_simplified(r);
5612        return r;
5613    }
5614
5615    /* Auxiliary method for `beta_reduce` and `beta_reduce_if_not_cases` */
5616    expr beta_reduce_cont(expr r, unsigned i, unsigned nargs, expr const *args,
5617                          bool is_let_val) {
5618        r = visit(r, false);
5619        if (i == nargs) return r;
5620        lean_assert(i < nargs);
5621        if (is_join_point_app(r)) {
5622            /* Expand join-point */
5623            lean_assert(!is_let_val);
5624            buffer<expr> new_args;
5625            expr const &jp = get_app_args(r, new_args);
5626            lean_assert(is_fvar(jp));
5627            for (; i < nargs; i++) new_args.push_back(args[i]);
5628            expr jp_val = *m_lctx.get_local_decl(jp).get_value();
5629            lean_assert(is_lambda(jp_val));
5630            return beta_reduce(jp_val, new_args.size(), new_args.data(), false);
5631        } else {
5632            if (!is_lcnf_atom(r)) r = mk_let_decl(r);
5633            return visit(mk_app(r, nargs - i, args + i), is_let_val);
5634        }
5635    }
5636
5637    expr beta_reduce(expr fn, unsigned nargs, expr const *args,
5638                     bool is_let_val) {
5639        unsigned i = 0;
5640        while (is_lambda(fn) && i < nargs) {
5641            i++;
5642            fn = binding_body(fn);
5643        }
5644        expr r = instantiate_rev(fn, i, args);
5645        if (is_lambda(r)) {
5646            lean_assert(i == nargs);
5647            return visit(r, is_let_val);
5648        } else {
5649            return beta_reduce_cont(r, i, nargs, args, is_let_val);
5650        }
5651    }
5652
5653    /* Remark: if `fn` is not a lambda expression, then this function
5654       will simply create the application `fn args_of(e)` */
5655    expr beta_reduce(expr fn, expr const &e, bool is_let_val) {
5656        buffer<expr> args;
5657        get_app_args(e, args);
5658        return beta_reduce(fn, args.size(), args.data(), is_let_val);
5659    }
5660
5661    bool should_inline_instance(name const &n) const {
5662        if (is_instance(env(), n))
5663            return !has_noinline_attribute(env(), n) &&
5664                   !has_init_attribute(env(), n);
5665        else
5666            return false;
5667    }
5668
5669    expr proj_constructor(expr const &k_app, unsigned proj_idx) {
```

```
5670         lean_assert(is_constructor_app(env(), k_app));
5671         buffer<expr> args;
5672         expr const &k = get_app_args(k_app, args);
5673         constructor_val k_val = env().get(const_name(k)).to_constructor_val();
5674         lean_assert(k_val.get_nparams() + proj_idx < args.size());
5675         return args[k_val.get_nparams() + proj_idx];
5676     }
5677
5678     optional<expr> try_inline_proj_instance_aux(expr s) {
5679         lean_assert(m_before_erasure);
5680         s = find(s);
5681         if (is_constructor_app(env(), s)) {
5682             return some_expr(s);
5683         } else if (is_proj(s)) {
5684             if (optional<expr> new_nested_s =
5685                     try_inline_proj_instance_aux(proj_expr(s))) {
5686                 lean_assert(is_constructor_app(env(), *new_nested_s));
5687                 expr r = proj_constructor(*new_nested_s,
5688                                           proj_idx(s).get_small_value());
5689                 return try_inline_proj_instance_aux(r);
5690             }
5691         } else {
5692             expr const &s_fn = get_app_fn(s);
5693             if (!is_constant(s_fn) || !should_inline_instance(const_name(s_fn)))
5694                 return none_expr();
5695             optional<constant_info> info =
5696                 env().find(mk_cstage1_name(const_name(s_fn)));
5697             if (!info || !info->is_definition()) return none_expr();
5698             if (get_app_num_args(s) < get_num_nested_lambdas(info->get_value()))
5699                 return none_expr();
5700             expr new_s_fn =
5701                 instantiate_value_lparams(*info, const_levels(s_fn));
5702             expr r = find(beta_reduce(new_s_fn, s, false));
5703             if (is_constructor_app(env(), r)) {
5704                 return some_expr(r);
5705             } else if (optional<expr> new_r = try_inline_proj_instance_aux(r)) {
5706                 return new_r;
5707             }
5708         }
5709         return none_expr();
5710     }
5711
5712     bool is_type_class(expr type) {
5713         type = cheap_beta_reduce(type);
5714         expr const &fn = get_app_fn(type);
5715         if (!is_constant(fn)) return false;
5716         return is_class(env(), const_name(fn));
5717     }
5718
5719     /* Auxiliary function for projecting "type class dictionary access".
5720        That is, we are trying to extract one of the type class instance
5721        elements.
5722
5723        Remark: We do not consider parent instances to be elements.
5724        For example, suppose `e` is `_x_4.1`, and we have
5725        ```
5726        _x_2 : Monad (ReaderT Bool (ExceptT String Id)) := @ReaderT.Monad Bool
5727        (ExceptT String Id) _x_1, _x_3 : Applicative (ReaderT Bool (ExceptT
5728        String Id)) := _x_2.1 _x_4 : Functor (ReaderT Bool (ExceptT String Id))
5729        := _x_3.1
5730        ```
5731        Then, we will expand `_x_4.1` since it corresponds to the `Functor` `map`
5732        element, and its type is not a type class, but is of the form
5733        ```
5734        (Π {α β : Type u}, (α → β) → ...)
5735        ```
5736        In the example above, the compiler should not expand `_x_3.1` or `_x_2.1`
5737        since their types type class applications: `Functor` and `Applicative`
5738        respectively. By eagerly expanding them, we may produce inefficient and
5739        bloated code. For example, we may be using `_x_3.1` to invoke a function
```

```
5740        that expects a `Functor` instance. By expanding `_x_3.1` we will be just
5741        expanding the code that creates this instance.
5742     */
5743     optional<expr> try_inline_proj_instance(expr const &e, bool is_let_val) {
5744         lean_assert(is_proj(e));
5745         if (!m_before_erasure) return none_expr();
5746         try {
5747             expr e_type = infer_type(e);
5748             if (is_type_class(e_type)) {
5749                 /* If `typeof(e)` is a type class, then we should not
5750                    instantiate it. See comment above. */
5751                 return none_expr();
5752             }
5753
5754             unsigned saved_fvars_size = m_fvars.size();
5755             if (optional<expr> new_s =
5756                     try_inline_proj_instance_aux(proj_expr(e))) {
5757                 lean_assert(is_constructor_app(env(), *new_s));
5758                 expr r =
5759                     proj_constructor(*new_s, proj_idx(e).get_small_value());
5760                 return some_expr(visit(r, is_let_val));
5761             }
5762             m_fvars.resize(saved_fvars_size);
5763             return none_expr();
5764         } catch (kernel_exception &) {
5765             return none_expr();
5766         }
5767     }
5768
5769     /* Return true iff `e` is of the form `fun (xs), let ys := ts in (ctor
5770        ...)`. This auxiliary method is used at try_inline_proj_instance_aux.
5771        It is a "quick" filter. */
5772     bool inline_proj_app_candidate(expr e) {
5773         while (is_lambda(e)) e = binding_body(e);
5774         while (is_let(e)) e = let_body(e);
5775         return static_cast<bool>(is_constructor_app(env(), e));
5776     }
5777
5778     /*
5779       Given `let x := f as in ... x.i`, where where `f` is defined as
5780       ```
5781       def f (xs) :=
5782       ...
5783       let y_i := t[xs] in
5784       ...
5785       ctor ... y_i ...
5786       ```
5787       reduce `x.i` into `t[as]`.
5788       `y_i` may depend on other let-declarations, but we only inline if the
5789       number of let-decl dependencies is less than `m_inline_threshold`.
5790
5791       Remark: this transformation is only applied before erasure.
5792       Remark: this transformation complements eager lambda lifting,
5793       and has been designed to optimize code such as:
5794       ```
5795       def f (x : nat) : Pro (Nat -> Nat) (Nat -> Bool) :=
5796       ((fun y, <code1 using x y>), (fun z, <code2 using x z>))
5797       ```
5798       That is, `f` is "packing" functions in a structure and returning it.
5799       Now, consider the following application:
5800       ```
5801       (f a).1 b
5802       ```
5803       With eager lambda lifting, we transform `f` into
5804       ```
5805       def f._elambda_1 (x y) : Nat :=
5806       <code1 using x y>
5807       def f._elambda_2 (x z) : Bool :=
5808       <code2 using x z>
5809       def f (x : nat) : Pro (Nat -> Nat) (Nat -> Bool) :=
```

```
  (f._elambda_1 x, f._elambda_2 x)
  ```
  Then, with this transformation, we transform `(f a).1` into
  `f._elambda_1 a`, and then with application merge, we transform
  `(f a).1 b` into `f._elambda_1 a b`

  See additional comments at `eager_lambda_lifting.cpp` */
optional<expr> try_inline_proj_app(expr const &e, bool is_let_val) {
    lean_assert(is_proj(e));
    if (!m_before_erasure) return none_expr();
    if (!proj_idx(e).is_small()) return none_expr();
    unsigned idx = proj_idx(e).get_small_value();
    expr s = find(proj_expr(e));
    buffer<expr> s_args;
    expr const &s_fn = get_app_rev_args(s, s_args);
    if (!is_constant(s_fn)) return none_expr();
    if (has_init_attribute(env(), const_name(s_fn))) return none_expr();
    if (has_noinline_attribute(env(), const_name(s_fn))) return none_expr();
    optional<constant_info> info =
        env().find(mk_cstage1_name(const_name(s_fn)));
    if (!info || !info->is_definition()) return none_expr();
    if (s_args.size() < get_num_nested_lambdas(info->get_value()))
        return none_expr();
    if (!inline_proj_app_candidate(info->get_value())) return none_expr();
    expr s_val = instantiate_value_lparams(*info, const_levels(s_fn));
    s_val = apply_beta(s_val, s_args.size(), s_args.data());
    buffer<expr> fvars;
    while (is_let(s_val)) {
        name n = mk_let_name(let_name(s_val));
        expr new_type =
            instantiate_rev(let_type(s_val), fvars.size(), fvars.data());
        expr new_val =
            instantiate_rev(let_value(s_val), fvars.size(), fvars.data());
        expr new_fvar = m_lctx.mk_local_decl(ngen(), n, new_type, new_val);
        fvars.push_back(new_fvar);
        s_val = let_body(s_val);
    }
    s_val = instantiate_rev(s_val, fvars.size(), fvars.data());
    lean_assert(is_constructor_app(env(), s_val));
    buffer<expr> k_args;
    expr const &k = get_app_args(s_val, k_args);
    constructor_val k_val = env().get(const_name(k)).to_constructor_val();
    lean_assert(k_val.get_nparams() + idx < k_args.size());
    expr val = k_args[k_val.get_nparams() + idx];
    buffer<expr> fvars_to_keep;
    name_hash_set used_fvars; /* Set of free variables names used */
    collect_used(val, used_fvars);
    unsigned i = fvars.size();
    while (i > 0) {
        i--;
        expr x = fvars[i];
        if (used_fvars.find(fvar_name(x)) != used_fvars.end()) {
            local_decl x_decl = m_lctx.get_local_decl(x);
            expr x_type = x_decl.get_type();
            expr x_val = *x_decl.get_value();
            collect_used(x_type, used_fvars);
            collect_used(x_val, used_fvars);
            fvars_to_keep.push_back(x);
            if (fvars_to_keep.size() > m_cfg.m_inline_threshold)
                return none_expr();
        }
    }
    std::reverse(fvars_to_keep.begin(), fvars_to_keep.end());
    val = m_lctx.mk_lambda(fvars_to_keep, val);
    return some_expr(visit(val, is_let_val));
}

expr visit_proj(expr const &e, bool is_let_val) {
    expr s = find_ctor(proj_expr(e));

```
```

```
5880         if (is_constructor_app(env(), s)) {
5881             return proj_constructor(s, proj_idx(e).get_small_value());
5882         }
5883
5884         if (optional<expr> r = try_inline_proj_instance(e, is_let_val)) {
5885             return *r;
5886         }
5887
5888         if (optional<expr> r = try_inline_proj_app(e, is_let_val)) {
5889             return *r;
5890         }
5891
5892         expr new_arg = visit_arg(proj_expr(e));
5893         if (is_eqp(proj_expr(e), new_arg))
5894             return e;
5895         else
5896             return update_proj(e, new_arg);
5897     }
5898
5899     expr reduce_cases_cnstr(buffer<expr> const &args,
5900                             inductive_val const &I_val, expr const &major,
5901                             bool is_let_val) {
5902         lean_assert(is_constructor_app(env(), major));
5903         unsigned nparams = I_val.get_nparams();
5904         buffer<expr> k_args;
5905         expr const &k = get_app_args(major, k_args);
5906         lean_assert(is_constant(k));
5907         lean_assert(nparams <= k_args.size());
5908         unsigned first_minor_idx = m_before_erasure
5909                                        ? (nparams + 1 /* typeformer/motive */ +
5910                                           I_val.get_nindices() + 1 /* major */)
5911                                        : 1;
5912         constructor_val k_val = env().get(const_name(k)).to_constructor_val();
5913         expr const &minor = args[first_minor_idx + k_val.get_cidx()];
5914         return beta_reduce(minor, k_args.size() - nparams,
5915                            k_args.data() + nparams, is_let_val);
5916     }
5917
5918     /* Just simplify minor premises. */
5919     expr visit_cases_default(expr const &e) {
5920         if (already_simplified(e)) return e;
5921         lean_assert(is_cases_on_app(env(), e));
5922         buffer<expr> args;
5923         expr const &c = get_app_args(e, args);
5924         /* simplify minor premises */
5925         bool all_equal_opt = true;
5926         optional<expr> a_minor;
5927         unsigned minor_idx;
5928         unsigned minors_end;
5929         std::tie(minor_idx, minors_end) =
5930             get_cases_on_minors_range(env(), const_name(c), m_before_erasure);
5931         expr const &major = args[minor_idx - 1];
5932         for (unsigned cidx = 0; minor_idx < minors_end; minor_idx++, cidx++) {
5933             expr minor = args[minor_idx];
5934             unsigned saved_fvars_size = m_fvars.size();
5935             buffer<expr> zs;
5936             minor = get_minor_body(minor, zs);
5937             expr new_minor;
5938             {
5939                 flet<expr2ctor> save_expr2ctor(m_expr2ctor, m_expr2ctor);
5940                 update_expr2ctor(major, c, args, cidx, zs);
5941                 new_minor = visit(minor, false);
5942             }
5943             new_minor = mk_let(zs, saved_fvars_size, new_minor, false);
5944             expr result_minor = mk_minor_lambda(zs, new_minor);
5945             if (all_equal_opt) {
5946                 expr result_minor_body = result_minor;
5947                 for (unsigned i = 0; i < zs.size(); i++) {
5948                     result_minor_body = binding_body(result_minor_body);
5949                     if (has_loose_bvars(result_minor_body)) {
```

```
5950                        /* Minor premise depends on constructor fields. */
5951                        all_equal_opt = false;
5952                        break;
5953                    }
5954                }
5955            }
5956            if (all_equal_opt) {
5957                if (!a_minor) {
5958                    a_minor = new_minor;
5959                } else if (new_minor != *a_minor) {
5960                    all_equal_opt = false;
5961                }
5962            }
5963            args[minor_idx] = result_minor;
5964        }
5965        if (all_equal_opt && a_minor && !is_join_point_app(*a_minor)) {
5966            /*
5967               Remark: we must make sure `a_minor` is not a joint-point.
5968               Otherwise, we would break our joint point application invariant.
5969               In the current implementation, this may seen as a hack or
5970               temporary workaround. Since the joint point inside of a
5971               non-terminal casesOn should not be allowed in the first place.
5972               When we reimplement this module in Lean, we should make sure this
5973               kind of term is not created by previous steps.
5974            */
5975            return *a_minor;
5976        }
5977        expr r = mk_app(c, args);
5978        mark_simplified(r);
5979        return r;
5980    }
5981
5982    /* Applies `Bool.casesOn x false true` ==> `x`
5983
5984       This transformation is often applicable to code that goes back and forth
5985       between `Decidable` and `Bool`.
5986       After `erase_irrelevant` both are `Bool`. */
5987    optional<expr> is_identity_bool_cases_on(inductive_val const &I_val,
5988                                             buffer<expr> const &args) {
5989        if (m_before_erasure) return none_expr();
5990        if (args.size() == 3 &&
5991            I_val.to_constant_val().get_name() == get_bool_name() &&
5992            args[1] == mk_bool_false() && args[2] == mk_bool_true()) {
5993            return some_expr(args[0]);
5994        }
5995        return none_expr();
5996    }
5997
5998    expr visit_cases(expr const &e, bool is_let_val) {
5999        buffer<expr> args;
6000        expr const &c = get_app_args(e, args);
6001        lean_assert(is_constant(c));
6002        inductive_val I_val = get_cases_on_inductive_val(env(), c);
6003        unsigned major_idx =
6004            get_cases_on_major_idx(env(), const_name(c), m_before_erasure);
6005        lean_assert(major_idx < args.size());
6006        expr major = find_ctor(args[major_idx]);
6007
6008        if (is_nat_lit(major)) {
6009            major = nat_lit_to_constructor(major);
6010        }
6011
6012        if (optional<expr> r = is_identity_bool_cases_on(I_val, args)) {
6013            return *r;
6014        }
6015
6016        if (is_constructor_app(env(), major)) {
6017            return reduce_cases_cnstr(args, I_val, major, is_let_val);
6018        } else if (!is_let_val) {
6019            return visit_cases_default(e);
```

```
        } else {
            return e;
        }
    }

    expr merge_app_app(expr const &fn, expr const &e, bool is_let_val) {
        lean_assert(is_app(fn));
        lean_assert(is_eqp(find(get_app_fn(e)), fn));
        lean_assert(!is_join_point_app(fn));
        if (!is_cases_on_app(env(), fn)) {
            buffer<expr> args;
            get_app_args(e, args);
            return visit_app(mk_app(fn, args), is_let_val);
        } else {
            return e;
        }
    }

    struct is_recursive_fn {
        environment const &m_env;
        csimp_cfg const &m_cfg;
        bool m_before_erasure;
        name m_target;

        is_recursive_fn(environment const &env, csimp_cfg const &cfg,
                        bool before_erasure)
            : m_env(env), m_cfg(cfg), m_before_erasure(before_erasure) {}

        optional<constant_info> is_inline_candidate(name const &f) {
            name c = m_before_erasure ? mk_cstage1_name(f) : mk_cstage2_name(f);
            optional<constant_info> info = m_env.find(c);
            if (!info || !info->is_definition()) {
                return optional<constant_info>();
            } else if (has_inline_attribute(m_env, f)) {
                return info;
            } else if (get_lcnf_size(m_env, info->get_value()) <=
                       m_cfg.m_inline_threshold) {
                return info;
            } else {
                return optional<constant_info>();
            }
        }

        bool visit(name const &f, name_set visited) {
            if (optional<constant_info> info = is_inline_candidate(f)) {
                if (visited.contains(f)) return true;
                visited.insert(f);
                return static_cast<bool>(::lean::find(
                    info->get_value(), [&](expr const &e, unsigned) {
                        return is_constant(e) &&
                               (const_name(e) == m_target ||
                                visit(const_name(e), visited));
                    }));
            } else {
                return false;
            }
        }

        bool operator()(name const &f) {
            m_target = f;
            return visit(f, name_set());
        }
    };

    /* We don't inline recursive functions. */
    bool is_recursive(name const &c) {
        return is_recursive_fn(env(), m_cfg, m_before_erasure)(c);
    }

    bool uses_unsafe_inductive(name const &c) {
```

```
6090         constant_info info = env().get(c);
6091         return static_cast<bool>(
6092             ::lean::find(info.get_value(), [&](expr const &e, unsigned) {
6093                 if (!is_constant(e) ||
6094                     !is_cases_on_recursor(env(), const_name(e)))
6095                     return false;
6096                 name const &I = const_name(e).get_prefix();
6097                 constant_info I_cinfo = env().get(I);
6098                 return I_cinfo.is_unsafe();
6099             }));
6100     }
6101
6102     bool is_stuck_at_cases(expr e) {
6103         type_checker tc(m_st, m_lctx);
6104         while (true) {
6105             bool cheap = true;
6106             expr e1 = tc.whnf_core(e, cheap);
6107             expr const &fn = get_app_fn(e1);
6108             if (!is_constant(fn)) return false;
6109             if (is_recursor(env(), const_name(fn))) return true;
6110             if (!is_cases_on_recursor(env(), const_name(fn))) return false;
6111             auto next_e = tc.unfold_definition(e1);
6112             if (!next_e) return true;
6113             e = *next_e;
6114         }
6115     }
6116
6117     optional<expr> beta_reduce_if_not_cases(expr fn, unsigned nargs,
6118                                             expr const *args, bool is_let_val) {
6119         unsigned i = 0;
6120         while (is_lambda(fn) && i < nargs) {
6121             i++;
6122             fn = binding_body(fn);
6123         }
6124         expr r = instantiate_rev(fn, i, args);
6125         if (is_lambda(r) || is_stuck_at_cases(r)) return none_expr();
6126         return some_expr(beta_reduce_cont(r, i, nargs, args, is_let_val));
6127     }
6128
6129     /* Auxiliary method used to inline functions marked with
6130        `[inline_if_reduce]`. It is similar to `beta_reduce`
6131        but it fails if the head is a `cases_on` application after `whnf_core`.
6132      */
6133     optional<expr> beta_reduce_if_not_cases(expr fn, expr const &e,
6134                                             bool is_let_val) {
6135         buffer<expr> args;
6136         get_app_args(e, args);
6137         return beta_reduce_if_not_cases(fn, args.size(), args.data(),
6138                                         is_let_val);
6139     }
6140
6141     bool check_noinline_attribute(name const &n) {
6142         if (!has_noinline_attribute(env(), n)) return false;
6143         /* Even if the function has `@[noinline]` attribute, we must still
6144            inline if its arguments were reduced by `reduce_arity`. This should
6145            only be checked after erasure. */
6146         if (m_before_erasure) return true;
6147         name c = mk_cstage2_name(n);
6148         optional<constant_info> info = env().find(c);
6149         if (!info || !info->is_definition()) return true;
6150         return !arity_was_reduced(comp_decl(n, info->get_value()));
6151     }
6152
6153     optional<expr> try_inline(expr const &fn, expr const &e, bool is_let_val) {
6154         lean_assert(is_constant(fn));
6155         lean_assert(is_constant(e) || is_eqp(find(get_app_fn(e)), fn));
6156         if (!m_cfg.m_inline) return none_expr();
6157         if (has_init_attribute(env(), const_name(fn))) return none_expr();
6158         if (check_noinline_attribute(const_name(fn))) return none_expr();
6159         if (m_before_erasure) {
```

```
6160            if (already_simplified(e)) return none_expr();
6161            name c = mk_cstage1_name(const_name(fn));
6162            optional<constant_info> info = env().find(c);
6163            if (!info || !info->is_definition()) return none_expr();
6164            if (get_app_num_args(e) < get_num_nested_lambdas(info->get_value()))
6165                return none_expr();
6166            bool inline_attr = has_inline_attribute(env(), const_name(fn));
6167            bool inline_if_reduce_attr =
6168                has_inline_if_reduce_attribute(env(), const_name(fn));
6169            if (!inline_attr && !inline_if_reduce_attr &&
6170                (get_lcnf_size(env(), info->get_value()) >
6171                     m_cfg.m_inline_threshold ||
6172                 is_constant(
6173                     e))) { /* We only inline constants if they are marked with
6174                               the `[inline]` or `[inline_if_reduce]` attrs */
6175                return none_expr();
6176            }
6177            if (!inline_if_reduce_attr && is_recursive(const_name(fn)))
6178                return none_expr();
6179            if (uses_unsafe_inductive(c)) return none_expr();
6180            expr new_fn = instantiate_value_lparams(*info, const_levels(fn));
6181            if (inline_if_reduce_attr && !inline_attr) {
6182                return beta_reduce_if_not_cases(new_fn, e, is_let_val);
6183            } else {
6184                return some_expr(beta_reduce(new_fn, e, is_let_val));
6185            }
6186        } else {
6187            /* We should not inline closed constants we have extracted. */
6188            if (is_extract_closed_aux_fn(const_name(fn))) return none_expr();
6189            name c = mk_cstage2_name(const_name(fn));
6190            optional<constant_info> info = env().find(c);
6191            if (!info || !info->is_definition()) return none_expr();
6192            unsigned arity = get_num_nested_lambdas(info->get_value());
6193            if (get_app_num_args(e) < arity || arity == 0) return none_expr();
6194            if (get_lcnf_size(env(), info->get_value()) >
6195                m_cfg.m_inline_threshold)
6196                return none_expr();
6197            if (is_recursive(const_name(fn))) return none_expr();
6198            if (uses_unsafe_inductive(c)) return none_expr();
6199            return some_expr(beta_reduce(info->get_value(), e, is_let_val));
6200        }
6201    }
6202
6203    expr visit_inline_app(expr const &e, bool is_let_val) {
6204        buffer<expr> args;
6205        get_app_args(e, args);
6206        lean_assert(!args.empty());
6207        if (args.size() < 2) return visit_app_default(e);
6208        buffer<expr> new_args;
6209        expr fn = get_app_args(find(args[1]), new_args);
6210        new_args.append(args.size() - 2, args.data() + 2);
6211        expr r = mk_app(fn, new_args);
6212        if (!m_cfg.m_inline || !is_constant(fn)) return visit(r, is_let_val);
6213        name main = const_name(fn);
6214        bool first = true;
6215        while (true) {
6216            name c = mk_cstage1_name(const_name(fn));
6217            optional<constant_info> info = env().find(c);
6218            if (!info || !info->is_definition())
6219                return first ? visit(r, is_let_val) : r;
6220            expr new_fn = instantiate_value_lparams(*info, const_levels(fn));
6221            r = beta_reduce(new_fn, new_args.size(), new_args.data(),
6222                            is_let_val);
6223            if (!is_app(r)) return r;
6224            fn = get_app_fn(r);
6225            /* If `r` is an application of the form `g ...` where
6226               `g` is an interal name and `g` prefix of the main function, we
6227               unfold this application too. */
6228            if (!is_constant(fn) || !is_internal_name(const_name(fn)) ||
6229                const_name(fn).get_prefix() != main)
```

```
6230                 return r;
6231             new_args.clear();
6232             get_app_args(r, new_args);
6233             first = false;
6234         }
6235     }
6236
6237     expr visit_app_default(expr const &e) {
6238         if (already_simplified(e)) return e;
6239         buffer<expr> args;
6240         bool modified = true;
6241         expr const &fn = get_app_args(e, args);
6242         for (expr &arg : args) {
6243             expr new_arg = visit_arg(arg);
6244             if (!is_eqp(arg, new_arg)) modified = true;
6245             arg = new_arg;
6246         }
6247         expr new_e = modified ? mk_app(fn, args) : e;
6248         mark_simplified(new_e);
6249         return new_e;
6250     }
6251
6252     expr visit_nat_succ(expr const &e) {
6253         expr arg = visit(app_arg(e), false);
6254         return mk_app(mk_constant(get_nat_add_name()), arg,
6255                       mk_lit(literal(nat(1))));
6256     }
6257
6258     expr visit_thunk_get(expr const &e, bool is_let_val) {
6259         buffer<expr> args;
6260         expr fn = get_app_args(e, args);
6261         lean_assert(is_constant(fn, get_thunk_get_name()));
6262         if (args.size() != 2) return visit_app_default(e);
6263         expr mk = find(args[1]);
6264         if (!is_app_of(mk, get_thunk_mk_name(), 2)) return visit_app_default(e);
6265         // @Thunk.get _ (@Thunk.mk _ g) => g ()
6266         expr g = app_arg(mk);
6267         return visit(mk_app(g, mk_unit_mk()), is_let_val);
6268     }
6269
6270     /*
6271       Replace `fixCore<n> f a_1 ... a_m`
6272       with `fixCore<m> f a_1 ... a_m` whenever `n < m`.
6273       This optimization is for writing reusable/generic code. For
6274       example, we cannot write an efficient `rec_t` monad transformer
6275       without it because we don't know the arity of `m A` when we write `rec_t`.
6276       Remark: the runtime provides a small set of `fixCore<i>` implementations
6277       (`i in [1, 6]`). This methods does nothing if `m > 6`. */
6278     expr visit_fix_core(expr const &e, unsigned n) {
6279         if (m_before_erasure) return visit_app_default(e);
6280         buffer<expr> args;
6281         expr fn = get_app_args(e, args);
6282         lean_assert(is_constant(fn) && is_fix_core(const_name(fn)));
6283         unsigned arity =
6284             n + /* α_1 ... α_n Type arguments */
6285             1 + /* β : Type */
6286             1 + /* (base : α_1 → ... → α_n → β) */
6287             1 + /* (rec : (α_1 → ... → α_n → β) → α_1 → ... → α_n → β) */
6288             n; /* α_1 → ... → α_n */
6289         if (args.size() <= arity) return visit_app_default(e);
6290         /* This `fixCore<n>` application is an overapplication.
6291            The `fixCore<n>` is implemented by the runtime, and the result
6292            is a closure. This is bad for performance. We should
6293            replace it with `fixCore<m>` (if the runtime contains one) */
6294         unsigned num_extra = args.size() - arity;
6295         unsigned m = n + num_extra;
6296         optional<expr> fix_core_m = mk_enf_fix_core(m);
6297         if (!fix_core_m) return visit_app_default(e);
6298         buffer<expr> new_args;
6299         /* Add α_1 ... α_n and β */
```

```
6300         for (unsigned i = 0; i < m + 1; i++) {
6301             new_args.push_back(mk_enf_neutral());
6302         }
6303         /* `(base : α_1 → ... → α_n → β)` is not used in the runtime primitive.
6304            So, we replace it with a neutral value :) */
6305         new_args.push_back(mk_enf_neutral());
6306         new_args.append(args.size() - n - 2, args.data() + n + 2);
6307         return mk_app(*fix_core_m, new_args);
6308     }
6309
6310     expr visit_app(expr const &e, bool is_let_val) {
6311         if (is_cases_on_app(env(), e)) {
6312             return visit_cases(e, is_let_val);
6313         } else if (is_app_of(e, get_inline_name())) {
6314             return visit_inline_app(e, is_let_val);
6315         }
6316         expr fn = find(get_app_fn(e));
6317         if (is_lambda(fn)) {
6318             return beta_reduce(fn, e, is_let_val);
6319         } else if (is_cases_on_app(env(), fn)) {
6320             expr new_e = float_cases_on_core(get_app_fn(e), e, fn);
6321             mark_simplified(new_e);
6322             return new_e;
6323         } else if (is_lc_unreachable_app(fn)) {
6324             lean_assert(m_before_erasure);
6325             expr type = infer_type(e);
6326             return mk_lc_unreachable(m_st, m_lctx, type);
6327         } else if (is_app(fn)) {
6328             return merge_app_app(fn, e, is_let_val);
6329         } else if (is_constant(fn)) {
6330             unsigned nargs = get_app_num_args(e);
6331             if (nargs == 1) {
6332                 expr a1 = find(visit_arg(app_arg(e)));
6333                 if (optional<expr> r = fold_un_op(m_before_erasure, fn, a1)) {
6334                     return *r;
6335                 }
6336             } else if (nargs == 2) {
6337                 expr a1 = find(visit_arg(app_arg(app_fn(e))));
6338                 expr a2 = find(visit_arg(app_arg(e)));
6339                 if (optional<expr> r =
6340                         fold_bin_op(m_before_erasure, fn, a1, a2)) {
6341                     return *r;
6342                 }
6343             }
6344             name const &n = const_name(fn);
6345             if (n == get_nat_succ_name()) {
6346                 return visit_nat_succ(e);
6347             } else if (n == get_nat_zero_name()) {
6348                 return mk_lit(literal(nat(0)));
6349             } else if (n == get_thunk_get_name()) {
6350                 return visit_thunk_get(e, is_let_val);
6351             } else if (optional<expr> r = try_inline(fn, e, is_let_val)) {
6352                 return *r;
6353             } else if (optional<unsigned> i = is_fix_core(n)) {
6354                 return visit_fix_core(e, *i);
6355             } else {
6356                 return visit_app_default(e);
6357             }
6358         } else {
6359             return visit_app_default(e);
6360         }
6361     }
6362
6363     expr visit_constant(expr const &e, bool is_let_val) {
6364         if (optional<expr> r = try_inline(e, e, is_let_val))
6365             return *r;
6366         else
6367             return e;
6368     }
6369
```

```
6370     expr visit_arg(expr const &e) {
6371         if (!is_lcnf_atom(e)) {
6372             /* non-atomic arguments are irrelevant in LCNF */
6373             return e;
6374         }
6375         expr new_e = visit(e, false);
6376         if (is_lcnf_atom(new_e))
6377             return new_e;
6378         else
6379             return mk_let_decl(new_e);
6380     }
6381
6382     expr visit(expr const &e, bool is_let_val) {
6383         switch (e.kind()) {
6384             case expr_kind::Lambda:
6385                 return is_let_val ? e : visit_lambda(e, false, false);
6386             case expr_kind::Let:
6387                 return visit_let(e);
6388             case expr_kind::Proj:
6389                 return visit_proj(e, is_let_val);
6390             case expr_kind::App:
6391                 return visit_app(e, is_let_val);
6392             case expr_kind::Const:
6393                 return visit_constant(e, is_let_val);
6394             default:
6395                 return e;
6396         }
6397     }
6398
6399    public:
6400     csimp_fn(environment const &env, local_ctx const &lctx, bool before_erasure,
6401              csimp_cfg const &cfg)
6402         : m_st(env),
6403           m_lctx(lctx),
6404           m_before_erasure(before_erasure),
6405           m_cfg(cfg),
6406           m_x("_x"),
6407           m_j("j") {}
6408
6409     expr operator()(expr const &e) {
6410         if (is_lambda(e)) {
6411             return visit_lambda(e, false, true);
6412         } else {
6413             buffer<expr> empty_xs;
6414             expr r = visit(e, false);
6415             return mk_let(empty_xs, 0, r, true);
6416         }
6417     }
6418 };
6419
6420 extern "C" uint8 lean_at_most_once(obj_arg e, obj_arg x);
6421
6422 bool at_most_once(expr const &e, name const &x) {
6423     inc_ref(e.raw());
6424     inc_ref(x.raw());
6425     return lean_at_most_once(e.raw(), x.raw());
6426 }
6427
6428 /* Eliminate join-points that are used only once */
6429 class elim_jp1_fn {
6430     environment const &m_env;
6431     local_ctx m_lctx;
6432     bool m_before_erasure;
6433     name_generator m_ngen;
6434     name_set m_to_expand;
6435     bool m_expanded{false};
6436
6437     void mark_to_expand(expr const &e) { m_to_expand.insert(fvar_name(e)); }
6438
6439     bool is_to_expand_jp_app(expr const &e) {
```

```
6440         expr const &f = get_app_fn(e);
6441         return is_fvar(f) && m_to_expand.contains(fvar_name(f));
6442     }
6443
6444     expr visit_lambda(expr e) {
6445         buffer<expr> fvars;
6446         while (is_lambda(e)) {
6447             expr domain = visit(
6448                 instantiate_rev(binding_domain(e), fvars.size(), fvars.data()));
6449             expr fvar = m_lctx.mk_local_decl(m_ngen, binding_name(e), domain,
6450                                              binding_info(e));
6451             fvars.push_back(fvar);
6452             e = binding_body(e);
6453         }
6454         e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
6455         return m_lctx.mk_lambda(fvars, e);
6456     }
6457
6458     expr visit_cases(expr const &e) {
6459         lean_assert(is_cases_on_app(m_env, e));
6460         buffer<expr> args;
6461         expr const &c = get_app_args(e, args);
6462         /* simplify minor premises */
6463         unsigned minor_idx;
6464         unsigned minors_end;
6465         std::tie(minor_idx, minors_end) =
6466             get_cases_on_minors_range(m_env, const_name(c), m_before_erasure);
6467         for (; minor_idx < minors_end; minor_idx++) {
6468             args[minor_idx] = visit(args[minor_idx]);
6469         }
6470         return mk_app(c, args);
6471     }
6472
6473     expr visit_app(expr const &e) {
6474         lean_assert(is_app(e));
6475         if (is_cases_on_app(m_env, e)) {
6476             return visit_cases(e);
6477         } else if (is_to_expand_jp_app(e)) {
6478             buffer<expr> args;
6479             expr const &jp = get_app_rev_args(e, args);
6480             local_decl jp_decl = m_lctx.get_local_decl(jp);
6481             lean_assert(is_join_point_name(jp_decl.get_user_name()));
6482             lean_assert(jp_decl.get_value());
6483             lean_assert(is_lambda(*jp_decl.get_value()));
6484             return apply_beta(*jp_decl.get_value(), args.size(), args.data());
6485         } else {
6486             return e;
6487         }
6488     }
6489
6490     bool at_most_once(expr const &e, expr const &jp) {
6491         lean_assert(is_fvar(jp));
6492         return lean::at_most_once(e, fvar_name(jp));
6493     }
6494
6495     expr visit_let(expr e) {
6496         buffer<expr> fvars;
6497         buffer<expr> all_fvars;
6498         while (is_let(e)) {
6499             expr new_type =
6500                 visit(instantiate_rev(let_type(e), fvars.size(), fvars.data()));
6501             expr new_val = visit(
6502                 instantiate_rev(let_value(e), fvars.size(), fvars.data()));
6503             expr fvar =
6504                 m_lctx.mk_local_decl(m_ngen, let_name(e), new_type, new_val);
6505             fvars.push_back(fvar);
6506             if (is_join_point_name(let_name(e))) {
6507                 e = instantiate_rev(let_body(e), fvars.size(), fvars.data());
6508                 fvars.clear();
6509                 if (at_most_once(e, fvar)) {
```

```
6510                     m_expanded = true;
6511                     mark_to_expand(fvar);
6512                 } else {
6513                     /* Keep join point */
6514                     all_fvars.push_back(fvar);
6515                 }
6516             } else {
6517                 all_fvars.push_back(fvar);
6518                 e = let_body(e);
6519             }
6520         }
6521         e = instantiate_rev(e, fvars.size(), fvars.data());
6522         e = visit(e);
6523         return m_lctx.mk_lambda(all_fvars, e);
6524     }
6525
6526     expr visit(expr const &e) {
6527         switch (e.kind()) {
6528             case expr_kind::Lambda:
6529                 return visit_lambda(e);
6530             case expr_kind::Let:
6531                 return visit_let(e);
6532             case expr_kind::App:
6533                 return visit_app(e);
6534             default:
6535                 return e;
6536         }
6537     }
6538
6539    public:
6540     elim_jp1_fn(environment const &env, local_ctx const &lctx,
6541                 bool before_erasure)
6542         : m_env(env), m_lctx(lctx), m_before_erasure(before_erasure) {}
6543     expr operator()(expr const &e) {
6544         m_expanded = false;
6545         return visit(e);
6546     }
6547
6548     bool expanded() const { return m_expanded; }
6549 };
6550
6551 expr csimp_core(environment const &env, local_ctx const &lctx, expr const &e0,
6552                 bool before_erasure, csimp_cfg const &cfg) {
6553     csimp_fn simp(env, lctx, before_erasure, cfg);
6554     elim_jp1_fn elim_jp1(env, lctx, before_erasure);
6555     expr e = e0;
6556     while (true) {
6557         e = simp(e);
6558         bool modified = false;
6559         e = elim_jp1(e);
6560         if (elim_jp1.expanded()) modified = true;
6561         expr new_e = cse_core(env, e, before_erasure);
6562         new_e = elim_dead_let(new_e);
6563         if (e != new_e) modified = true;
6564         if (!modified) return e;
6565         e = new_e;
6566     }
6567 }
6568 }  // namespace lean
6569 // ::::::::::::::
6570 // :compiler/eager_lambda_lifting.cpp
6571 // ::::::::::::::
6572 /*
6573 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6574 Released under Apache 2.0 license as described in the file LICENSE.
6575
6576 Author: Leonardo de Moura
6577 */
6578 #include <lean/flet.h>
6579
```

```
#include "kernel/abstract.h"
#include "kernel/for_each_fn.h"
#include "kernel/instantiate.h"
#include "kernel/type_checker.h"
#include "library/class.h"
#include "library/compiler/closed_term_cache.h"
#include "library/compiler/csimp.h"
#include "library/compiler/util.h"
#include "library/trace.h"

namespace lean {
extern "C" object *lean_mk_eager_lambda_lifting_name(object *n, object *idx);
extern "C" uint8 lean_is_eager_lambda_lifting_name(object *n);

name mk_elambda_lifting_name(name const &fn, unsigned idx) {
    return name(
        lean_mk_eager_lambda_lifting_name(fn.to_obj_arg(), mk_nat_obj(idx)));
}

bool is_elambda_lifting_name(name fn) {
    return lean_is_eager_lambda_lifting_name(fn.to_obj_arg());
}

/* Return true iff `e` contains a free variable that is not in `exception_set`.
 */
static bool has_fvar_except(expr const &e, name_set const &exception_set) {
    if (!has_fvar(e)) return false;
    bool found = false;
    for_each(e, [&](expr const &e, unsigned) {
        if (!has_fvar(e)) return false;
        if (found) return false;  // done
        if (is_fvar(e) && !exception_set.contains(fvar_name(e))) {
            found = true;
            return false;  // done
        }
        return true;
    });
    return found;
}

/* Return true if the type of a parameter in `params` depends on `fvar`. */
static bool depends_on_fvar(local_ctx const &lctx, buffer<expr> const &params,
                            expr const &fvar) {
    for (expr const &param : params) {
        local_decl const &decl = lctx.get_local_decl(param);
        lean_assert(!decl.get_value());
        if (has_fvar(decl.get_type(), fvar)) return true;
    }
    return false;
}

/*
   We eagerly lift lambda expressions that are stored in terminal constructors.
   We say a constructor application is terminal if it is the result/returned.
   We use this transformation to generate good code for the following scenario:
   Suppose we have a definition
   ```
   def f (x : nat) : Pro (Nat -> Nat) (Nat -> Bool) :=
   ((fun y, <code1 using x y>), (fun z, <code2 using x z>))
   ```
   That is, `f` is "packing" functions in a structure and returning it.
   Now, consider the following application:
   ```
   (f a).1 b
   ```
   Without eager lambda lifting, `f a` will create two closures and one pair.
   Then, we project the first closure in the pair and apply it to `b`.
   This is inefficient. If `f` is small, we can workaround this problem by
   inlining `f`. However, if inlining is not feasible, we would have to perform
   all memory allocations. This is particularly bad, if `f` is a structure with
```

```
6650    many fields. With eager lambda lifting, we transform `f` into
6651    ```
6652    def f._elambda_1 (x y) : Nat :=
6653    <code1 using x y>
6654    def f._elambda_2 (x z) : Bool :=
6655    <code2 using x z>
6656    def f (x : nat) : Pro (Nat -> Nat) (Nat -> Bool) :=
6657    (f._elambda_1 x, f._elambda_2 x)
6658    ```
6659    Then, when the simplifier sees `(f a).1 b`, it can reduce it to `f._elambda_1
6660    a b`, and closure and pair allocations are avoided.
6661
6662    Note that we do not lift all nested lambdas here, only the ones in terminal
6663    constructors. Premature lambda lifting may hurt performance in the
6664    non-terminal case. Example:
6665    ```
6666    def f (xs : List Nat) :=
6667    let g := fun x, x + x in
6668    List.map g xs
6669    ```
6670    We want to keep `fun x, x+x` until we specialize `f`.
6671
6672    Remark: we also skip this transformation for definitions marked as `[inline]`
6673    or `[instance]`.
6674 */
6675 class eager_lambda_lifting_fn {
6676     type_checker::state m_st;
6677     csimp_cfg m_cfg;
6678     local_ctx m_lctx;
6679     buffer<comp_decl> m_new_decls;
6680     name m_base_name;
6681     name_set m_closed_fvars; /* let-declarations that only depend on global
6682                                 constants and other closed_fvars */
6683     name_set m_terminal_lambdas;
6684     name_set m_nonterminal_lambdas;
6685     unsigned m_next_idx{1};
6686
6687     environment const &env() const { return m_st.env(); }
6688
6689     name_generator &ngen() { return m_st.ngen(); }
6690
6691     expr eta_expand(expr const &e) { return lcnf_eta_expand(m_st, m_lctx, e); }
6692
6693     name next_name() {
6694         name r = mk_elambda_lifting_name(m_base_name, m_next_idx);
6695         m_next_idx++;
6696         return r;
6697     }
6698
6699     bool collect_fvars_core(expr const &e, name_set &collected,
6700                             buffer<expr> &fvars) {
6701         if (!has_fvar(e)) return true;
6702         bool ok = true;
6703         for_each(e, [&](expr const &x, unsigned) {
6704             if (!has_fvar(x)) return false;
6705             if (!ok) return false;
6706             if (is_fvar(x)) {
6707                 if (!collected.contains(fvar_name(x))) {
6708                     collected.insert(fvar_name(x));
6709                     local_decl d = m_lctx.get_local_decl(x);
6710                     /* We do not eagerly lift a lambda if we need to copy a
6711                        join-point. Remark: we may revise this decision in the
6712                        future, and use the same approach we use at
6713                        `lambda_lifting.cpp`.
6714                      */
6715                     if (is_join_point_name(d.get_user_name())) {
6716                         ok = false;
6717                         return false;
6718                     } else {
6719                         if (!collect_fvars_core(d.get_type(), collected, fvars))
```

```
                            return false;
                        if (m_closed_fvars.contains(fvar_name(x))) {
                            /* If x only depends on global constants and other
                               variables in m_closed_fvars. Then, we also
                               collect the other variables at m_closed_fvars. */
                            if (!collect_fvars_core(*d.get_value(), collected,
                                                    fvars))
                                return false;
                        }
                        fvars.push_back(x);
                    }
                }
            }
            return true;
        });
        return ok;
    }

    bool collect_fvars(expr const &e, buffer<expr> &fvars) {
        if (!has_fvar(e)) return true;
        name_set collected;
        if (collect_fvars_core(e, collected, fvars)) {
            sort_fvars(m_lctx, fvars);
            return true;
        } else {
            return false;
        }
    }

    /* Split fvars in two groups: `new_params` and `to_copy`.
       We put a fvar `x` in `new_params` if it is not a let declaration,
       or a variable in `params` depend on `x`, or it is not in
       `m_closed_fvars`.

       The variables in `to_copy` are variables that depend only on
       global constants or other variables in `to_copy`, and `params` do not
       depend on them. */
    void split_fvars(buffer<expr> const &fvars, buffer<expr> const &params,
                     buffer<expr> &new_params, buffer<expr> &to_copy) {
        for (expr const &fvar : fvars) {
            local_decl const &decl = m_lctx.get_local_decl(fvar);
            if (!decl.get_value()) {
                new_params.push_back(fvar);
            } else {
                if (!m_closed_fvars.contains(fvar_name(fvar)) ||
                    depends_on_fvar(m_lctx, params, fvar)) {
                    new_params.push_back(fvar);
                } else {
                    to_copy.push_back(fvar);
                }
            }
        }
    }

    expr lift_lambda(expr e, bool apply_simp) {
        /* Hack: We use `try` here because previous compilation steps may have
           produced type incorrect terms. */
        try {
            lean_assert(is_lambda(e));
            buffer<expr> fvars;
            if (!collect_fvars(e, fvars)) {
                return e;
            }
            buffer<expr> params;
            while (is_lambda(e)) {
                expr param_type = instantiate_rev(binding_domain(e),
                                                  params.size(), params.data());
                expr param = m_lctx.mk_local_decl(ngen(), binding_name(e),
                                                  param_type, binding_info(e));
                params.push_back(param);
```

```
6790                e = binding_body(e);
6791            }
6792            e = instantiate_rev(e, params.size(), params.data());
6793            buffer<expr> new_params, to_copy;
6794            split_fvars(fvars, params, new_params, to_copy);
6795            /*
6796              Variables in `to_copy` only depend on global constants
6797              and other variables in `to_copy`. Moreover, `params` do not depend
6798              on them. It is wasteful to pass them as new parameters to the new
6799              lifted declaration. We can just copy them. The code duplication is
6800              not problematic because later at `extract_closed` we will create
6801              global names for closed terms, and eliminate the redundancy.
6802            */
6803            e = m_lctx.mk_lambda(to_copy, e);
6804            e = m_lctx.mk_lambda(params, e);
6805            expr code = abstract(e, new_params.size(), new_params.data());
6806            unsigned i = new_params.size();
6807            while (i > 0) {
6808                --i;
6809                local_decl const &decl = m_lctx.get_local_decl(new_params[i]);
6810                expr type = abstract(decl.get_type(), i, new_params.data());
6811                code = ::lean::mk_lambda(decl.get_user_name(), type, code);
6812            }
6813            if (apply_simp) {
6814                code = csimp(env(), code, m_cfg);
6815            }
6816            expr type = cheap_beta_reduce(type_checker(m_st).infer(code));
6817            name n = next_name();
6818            /* We add the auxiliary declaration `n` as a "meta" axiom to the
6819               environment. This is a hack to make sure we can use `csimp` to
6820               simplify `code` and other definitions that use `n`. We used a
6821               similar hack at `specialize.cpp`. */
6822            declaration aux_ax = mk_axiom(n, names(), type, true /* meta */);
6823            m_st.env() = env().add(aux_ax, false);
6824            m_new_decls.push_back(comp_decl(n, code));
6825            return mk_app(mk_constant(n), new_params);
6826        } catch (exception &) {
6827            return e;
6828        }
6829    }
6830
6831    /* Given a free variable `x`, follow let-decls and return a pair `(x, v)`.
6832       Examples for `find(x)`
6833       - `x := 1` ==> `(x, 1)`
6834       - `z := (fun w, w+1); y := z; x := y` ==> `(z, (fun w, w+1))`
6835       - `z := f a; y := mdata kv z; x := y` ==> `(z, f a)`
6836    */
6837    pair<name, expr> find(expr const &x) const {
6838        lean_assert(is_fvar(x));
6839        expr e = x;
6840        name r = fvar_name(x);
6841        while (true) {
6842            if (is_mdata(e)) {
6843                e = mdata_expr(e);
6844            } else if (is_fvar(e)) {
6845                r = fvar_name(e);
6846                optional<local_decl> decl = m_lctx.find_local_decl(e);
6847                lean_assert(decl);
6848                if (optional<expr> v = decl->get_value()) {
6849                    if (is_join_point_name(decl->get_user_name())) {
6850                        return mk_pair(r, e);
6851                    } else {
6852                        e = *v;
6853                    }
6854                } else {
6855                    return mk_pair(r, e);
6856                }
6857            } else {
6858                return mk_pair(r, e);
6859            }
```

```
6860         }
6861     }
6862
6863     expr visit_lambda_core(expr e) {
6864         flet<local_ctx> save_lctx(m_lctx, m_lctx);
6865         buffer<expr> fvars;
6866         while (is_lambda(e)) {
6867             expr new_type =
6868                 instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
6869             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
6870                                                  new_type, binding_info(e));
6871             fvars.push_back(new_fvar);
6872             e = binding_body(e);
6873         }
6874         expr r = visit_terminal(instantiate_rev(e, fvars.size(), fvars.data()));
6875         return m_lctx.mk_lambda(fvars, r);
6876     }
6877
6878     expr visit_let(expr e) {
6879         flet<local_ctx> save_lctx(m_lctx, m_lctx);
6880         buffer<expr> fvars;
6881         while (is_let(e)) {
6882             bool not_root = false;
6883             bool jp = is_join_point_name(let_name(e));
6884             expr new_type =
6885                 instantiate_rev(let_type(e), fvars.size(), fvars.data());
6886             expr new_val =
6887                 visit(instantiate_rev(let_value(e), fvars.size(), fvars.data()),
6888                       not_root, jp);
6889             expr new_fvar =
6890                 m_lctx.mk_local_decl(ngen(), let_name(e), new_type, new_val);
6891             if (!has_fvar_except(new_type, m_closed_fvars) &&
6892                 !has_fvar_except(new_val, m_closed_fvars)) {
6893                 m_closed_fvars.insert(fvar_name(new_fvar));
6894             }
6895             fvars.push_back(new_fvar);
6896             e = let_body(e);
6897         }
6898         expr r = visit_terminal(instantiate_rev(e, fvars.size(), fvars.data()));
6899         r = abstract(r, fvars.size(), fvars.data());
6900         unsigned i = fvars.size();
6901         while (i > 0) {
6902             --i;
6903             name const &n = fvar_name(fvars[i]);
6904             local_decl const &decl = m_lctx.get_local_decl(n);
6905             expr type = abstract(decl.get_type(), i, fvars.data());
6906             expr val = *decl.get_value();
6907             if (m_terminal_lambdas.contains(n) &&
6908                 !m_nonterminal_lambdas.contains(n)) {
6909                 expr new_val = eta_expand(val);
6910                 lean_assert(is_lambda(new_val));
6911                 bool apply_simp = new_val != val;
6912                 val = lift_lambda(new_val, apply_simp);
6913             }
6914             r = ::lean::mk_let(decl.get_user_name(), type,
6915                                abstract(val, i, fvars.data()), r);
6916         }
6917         return r;
6918     }
6919
6920     expr visit_cases_on(expr const &e) {
6921         lean_assert(is_cases_on_app(env(), e));
6922         buffer<expr> args;
6923         expr const &c = get_app_args(e, args);
6924         /* Remark: eager lambda lifting is applied before we have erased most
6925          * type information. */
6926         unsigned minor_idx;
6927         unsigned minors_end;
6928         bool before_erasure = true;
6929         std::tie(minor_idx, minors_end) =
```

```
6930            get_cases_on_minors_range(env(), const_name(c), before_erasure);
6931        for (; minor_idx < minors_end; minor_idx++) {
6932            args[minor_idx] = visit_lambda_core(args[minor_idx]);
6933        }
6934        return mk_app(c, args);
6935    }
6936
6937    expr visit_app(expr const &e) {
6938        if (is_cases_on_app(env(), e)) {
6939            return visit_cases_on(e);
6940        } else {
6941            buffer<expr> args;
6942            get_app_args(e, args);
6943            for (expr const &arg : args) {
6944                if (is_fvar(arg)) {
6945                    name x;
6946                    expr v;
6947                    std::tie(x, v) = find(arg);
6948                    if (is_lambda(v)) {
6949                        m_nonterminal_lambdas.insert(x);
6950                    }
6951                }
6952            }
6953            return e;
6954        }
6955    }
6956
6957    expr visit_lambda(expr const &e, bool root, bool join_point) {
6958        if (root || join_point)
6959            return visit_lambda_core(e);
6960        else
6961            return e;
6962    }
6963
6964    expr visit(expr const &e, bool root = false, bool join_point = false) {
6965        switch (e.kind()) {
6966            case expr_kind::App:
6967                return visit_app(e);
6968            case expr_kind::Lambda:
6969                return visit_lambda(e, root, join_point);
6970            case expr_kind::Let:
6971                return visit_let(e);
6972            default:
6973                return e;
6974        }
6975    }
6976
6977    expr visit_terminal(expr const &e) {
6978        expr t = is_fvar(e) ? find(e).second : e;
6979        if (is_constructor_app(env(), t)) {
6980            buffer<expr> args;
6981            get_app_args(e, args);
6982            for (expr const &arg : args) {
6983                if (is_fvar(arg)) {
6984                    name x;
6985                    expr v;
6986                    std::tie(x, v) = find(arg);
6987                    v = eta_expand(v);
6988                    if (is_lambda(v)) {
6989                        m_terminal_lambdas.insert(x);
6990                    }
6991                }
6992            }
6993            return e;
6994        } else {
6995            return visit(e);
6996        }
6997    }
6998
6999    public:
```

```
7000     eager_lambda_lifting_fn(environment const &env, csimp_cfg const &cfg)
7001         : m_st(env), m_cfg(cfg) {}
7002
7003     pair<environment, comp_decls> operator()(comp_decl const &cdecl) {
7004         m_base_name = cdecl.fst();
7005         expr r = visit(cdecl.snd(), true);
7006         comp_decl new_cdecl(cdecl.fst(), r);
7007         m_new_decls.push_back(new_cdecl);
7008         return mk_pair(env(), comp_decls(m_new_decls));
7009     }
7010 };
7011
7012 pair<environment, comp_decls> eager_lambda_lifting(environment env,
7013                                                    comp_decls const &ds,
7014                                                    csimp_cfg const &cfg) {
7015     comp_decls r;
7016     for (comp_decl const &d : ds) {
7017         if (has_inline_attribute(env, d.fst()) || is_instance(env, d.fst())) {
7018             r = append(r, comp_decls(d));
7019         } else {
7020             comp_decls new_ds;
7021             std::tie(env, new_ds) = eager_lambda_lifting_fn(env, cfg)(d);
7022             r = append(r, new_ds);
7023         }
7024     }
7025     return mk_pair(env, r);
7026 }
7027 }  // namespace lean
7028 // ::::::::::::::
7029 // compiler/elim_dead_let.cpp
7030 // ::::::::::::::
7031 /*
7032 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
7033 Released under Apache 2.0 license as described in the file LICENSE.
7034
7035 Author: Leonardo de Moura
7036 */
7037 #include <algorithm>
7038 #include <unordered_set>
7039
7040 #include "kernel/abstract.h"
7041 #include "kernel/for_each_fn.h"
7042 #include "kernel/instantiate.h"
7043 #include "util/name_generator.h"
7044
7045 namespace lean {
7046 static name *g_elim_dead_let_fresh = nullptr;
7047
7048 class elim_dead_let_fn {
7049     std::unordered_set<name, name_hash_fn> m_used;
7050     name_generator m_ngen;
7051
7052     void mark_fvar(expr const &e) { m_used.insert(fvar_name(e)); }
7053
7054     expr visit_let(expr e) {
7055         buffer<expr> fvars;
7056         buffer<expr> lets;
7057         while (is_let(e)) {
7058             expr fvar = mk_fvar(m_ngen.next());
7059             fvars.push_back(fvar);
7060             lets.push_back(e);
7061             e = let_body(e);
7062         }
7063         e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
7064         buffer<expr> used;
7065         buffer<std::tuple<name, expr, expr>> entries;
7066         while (!fvars.empty()) {
7067             expr fvar = fvars.back();
7068             fvars.pop_back();
7069             expr let = lets.back();
```

```
7070            lets.pop_back();
7071            if (m_used.find(fvar_name(fvar)) != m_used.end()) {
7072                expr new_type = visit(
7073                    instantiate_rev(let_type(let), fvars.size(), fvars.data()));
7074                expr new_value = visit(instantiate_rev(
7075                    let_value(let), fvars.size(), fvars.data()));
7076                used.push_back(fvar);
7077                entries.emplace_back(let_name(let), new_type, new_value);
7078            }
7079        }
7080        std::reverse(used.begin(), used.end());
7081        std::reverse(entries.begin(), entries.end());
7082        e = abstract(e, used.size(), used.data());
7083        unsigned i = entries.size();
7084        while (i > 0) {
7085            --i;
7086            expr new_type = abstract(std::get<1>(entries[i]), i, used.data());
7087            expr new_value = abstract(std::get<2>(entries[i]), i, used.data());
7088            e = mk_let(std::get<0>(entries[i]), new_type, new_value, e);
7089        }
7090        return e;
7091    }
7092
7093    expr visit_lambda(expr e) {
7094        buffer<expr> fvars;
7095        buffer<std::tuple<name, expr, binder_info>> entries;
7096        while (is_lambda(e)) {
7097            expr domain = visit(
7098                instantiate_rev(binding_domain(e), fvars.size(), fvars.data()));
7099            expr fvar = mk_fvar(m_ngen.next());
7100            entries.emplace_back(binding_name(e), domain, binding_info(e));
7101            fvars.push_back(fvar);
7102            e = binding_body(e);
7103        }
7104        e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
7105        e = abstract(e, fvars.size(), fvars.data());
7106        unsigned i = entries.size();
7107        while (i > 0) {
7108            --i;
7109            expr new_domain =
7110                abstract(std::get<1>(entries[i]), i, fvars.data());
7111            e = mk_lambda(std::get<0>(entries[i]), new_domain, e,
7112                          std::get<2>(entries[i]));
7113        }
7114        return e;
7115    }
7116
7117    expr visit_app(expr const &e) {
7118        expr fn = visit(app_fn(e));
7119        expr arg = visit(app_arg(e));
7120        return update_app(e, fn, arg);
7121    }
7122
7123    expr visit_proj(expr const &e) {
7124        return update_proj(e, visit(proj_expr(e)));
7125    }
7126
7127    expr visit_mdata(expr const &e) {
7128        return update_mdata(e, visit(mdata_expr(e)));
7129    }
7130
7131    expr visit_pi(expr const &e) {
7132        for_each(e, [&](expr const &e, unsigned) {
7133            if (is_fvar(e)) mark_fvar(e);
7134            return true;
7135        });
7136        return e;
7137    }
7138
7139    expr visit(expr const &e) {
```

```
7140         switch (e.kind()) {
7141             case expr_kind::Lambda:
7142                 return visit_lambda(e);
7143             case expr_kind::Let:
7144                 return visit_let(e);
7145             case expr_kind::Proj:
7146                 return visit_proj(e);
7147             case expr_kind::App:
7148                 return visit_app(e);
7149             case expr_kind::FVar:
7150                 mark_fvar(e);
7151                 return e;
7152             case expr_kind::MData:
7153                 return visit_mdata(e);
7154             case expr_kind::Pi:
7155                 return visit_pi(e);
7156             case expr_kind::Const:
7157                 return e;
7158             case expr_kind::Sort:
7159                 return e;
7160             case expr_kind::Lit:
7161                 return e;
7162             case expr_kind::BVar:
7163                 return e;
7164             case expr_kind::MVar:
7165                 lean_unreachable();
7166         }
7167         lean_unreachable();
7168     }
7169
7170    public:
7171     elim_dead_let_fn() : m_ngen(*g_elim_dead_let_fresh) {}
7172
7173     expr operator()(expr const &e) { return visit(e); }
7174 };
7175
7176 expr elim_dead_let(expr const &e) { return elim_dead_let_fn()(e); }
7177
7178 void initialize_elim_dead_let() {
7179     g_elim_dead_let_fresh = new name("_elim_dead_let_fresh");
7180     mark_persistent(g_elim_dead_let_fresh->raw());
7181     register_name_generator_prefix(*g_elim_dead_let_fresh);
7182 }
7183 void finalize_elim_dead_let() { delete g_elim_dead_let_fresh; }
7184 }  // namespace lean
7185 // ::::::::::::::
7186 // compiler/erase_irrelevant.cpp
7187 // ::::::::::::::
7188 /*
7189 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
7190 Released under Apache 2.0 license as described in the file LICENSE.
7191
7192 Author: Leonardo de Moura
7193 */
7194 #include <lean/flet.h>
7195
7196 #include "kernel/abstract.h"
7197 #include "kernel/instantiate.h"
7198 #include "kernel/kernel_exception.h"
7199 #include "kernel/type_checker.h"
7200 #include "library/compiler/util.h"
7201
7202 namespace lean {
7203 class erase_irrelevant_fn {
7204     typedef std::tuple<name, expr, expr> let_entry;
7205     type_checker::state m_st;
7206     local_ctx m_lctx;
7207     buffer<expr> m_let_fvars;
7208     buffer<let_entry> m_let_entries;
7209     name m_x;
```

```
7210     unsigned m_next_idx{1};
7211     expr_map<bool> m_irrelevant_cache;
7212
7213     environment &env() { return m_st.env(); }
7214
7215     name_generator &ngen() { return m_st.ngen(); }
7216
7217     name next_name() {
7218         name r = m_x.append_after(m_next_idx);
7219         m_next_idx++;
7220         return r;
7221     }
7222
7223     expr infer_type(expr const &e) {
7224         return type_checker(m_st, m_lctx).infer(e);
7225     }
7226
7227     optional<unsigned> has_trivial_structure(name const &I_name) {
7228         return ::lean::has_trivial_structure(env(), I_name);
7229     }
7230
7231     expr mk_runtime_type(expr e) {
7232         return ::lean::mk_runtime_type(m_st, m_lctx, e);
7233     }
7234
7235     bool cache_is_irrelevant(expr const &e, bool r) {
7236         if (is_constant(e) || is_fvar(e))
7237             m_irrelevant_cache.insert(mk_pair(e, r));
7238         return r;
7239     }
7240
7241     bool is_irrelevant(expr const &e) {
7242         if (is_constant(e) || is_fvar(e)) {
7243             auto it1 = m_irrelevant_cache.find(e);
7244             if (it1 != m_irrelevant_cache.end()) return it1->second;
7245         }
7246         try {
7247             type_checker tc(m_st, m_lctx);
7248             expr type = tc.whnf(tc.infer(e));
7249             bool r = is_irrelevant_type(m_st, m_lctx, type);
7250             return cache_is_irrelevant(e, r);
7251         } catch (kernel_exception &) {
7252             /* failed to infer type or normalize, assume it is relevant */
7253             return cache_is_irrelevant(e, false);
7254         }
7255     }
7256
7257     expr visit_constant(expr const &e) {
7258         lean_assert(!is_enf_neutral(e));
7259         name const &c = const_name(e);
7260         if (c == get_lc_unreachable_name()) {
7261             return mk_enf_unreachable();
7262         } else if (c == get_lc_proof_name()) {
7263             return mk_enf_neutral();
7264         } else if (is_irrelevant(e)) {
7265             return mk_enf_neutral();
7266         } else {
7267             return mk_constant(const_name(e));
7268         }
7269     }
7270
7271     expr visit_fvar(expr const &e) {
7272         if (is_irrelevant(e)) {
7273             return mk_enf_neutral();
7274         } else {
7275             return e;
7276         }
7277     }
7278
7279     bool is_atom(expr const &e) {
```

```
7280         switch (e.kind()) {
7281             case expr_kind::FVar:
7282                 return true;
7283             case expr_kind::Lit:
7284                 return true;
7285             case expr_kind::Const:
7286                 return true;
7287             default:
7288                 return false;
7289         }
7290     }
7291
7292     expr visit_lambda_core(expr e, bool is_minor) {
7293         flet<local_ctx> save_lctx(m_lctx, m_lctx);
7294         buffer<expr> bfvars;
7295         buffer<pair<name, expr>> entries;
7296         while (is_lambda(e)) {
7297             /* Types are ignored in compilation steps. So, we do not invoke
7298              * visit for d. */
7299             expr d = instantiate_rev(binding_domain(e), bfvars.size(),
7300                                      bfvars.data());
7301             expr fvar = m_lctx.mk_local_decl(ngen(), binding_name(e), d,
7302                                              binding_info(e));
7303             bfvars.push_back(fvar);
7304             entries.emplace_back(binding_name(e), mk_runtime_type(d));
7305             e = binding_body(e);
7306         }
7307         unsigned saved_let_fvars_size = m_let_fvars.size();
7308         lean_assert(m_let_entries.size() == m_let_fvars.size());
7309         e = instantiate_rev(e, bfvars.size(), bfvars.data());
7310         if (is_irrelevant(e)) return mk_enf_neutral();
7311         expr r = visit(e);
7312         r = mk_let(saved_let_fvars_size, r);
7313         if (is_minor && is_lambda(r)) {
7314             /* Remark: we don't want to mix the lambda for minor premise fields,
7315              * with the result. */
7316             r = ::lean::mk_let("_x", mk_enf_object_type(), r, mk_bvar(0));
7317         }
7318         r = abstract(r, bfvars.size(), bfvars.data());
7319         unsigned i = entries.size();
7320         while (i > 0) {
7321             --i;
7322             r = mk_lambda(entries[i].first, entries[i].second, r);
7323         }
7324         return r;
7325     }
7326
7327     expr visit_lambda(expr const &e) { return visit_lambda_core(e, false); }
7328
7329     expr visit_minor(expr const &e) { return visit_lambda_core(e, true); }
7330
7331     expr mk_simple_decl(expr const &e, expr const &e_type) {
7332         name n = next_name();
7333         expr x = m_lctx.mk_local_decl(ngen(), n, e_type, e);
7334         m_let_fvars.push_back(x);
7335         m_let_entries.emplace_back(n, mk_runtime_type(e_type), e);
7336         return x;
7337     }
7338
7339     static expr mk_list_char() {
7340         return mk_app(mk_constant(get_list_name(), {mk_level_zero()}),
7341                       mk_constant(get_char_name()));
7342     }
7343
7344     expr elim_string_cases(buffer<expr> &args) {
7345         lean_assert(args.size() == 3);
7346         expr major = visit(args[1]);
7347         expr x = mk_simple_decl(
7348             mk_app(mk_constant(get_string_data_name()), major), mk_list_char());
7349         expr minor = args[2];
```

```
7350         minor = instantiate(binding_body(minor), x);
7351         return visit(minor);
7352     }
7353
7354     expr elim_nat_cases(buffer<expr> &args) {
7355         lean_assert(args.size() == 4);
7356         expr major = visit(args[1]);
7357         expr zero = mk_lit(literal(nat(0)));
7358         expr one = mk_lit(literal(nat(1)));
7359         expr nat_type = mk_constant(get_nat_name());
7360         expr dec_eq = mk_app(mk_constant(get_nat_dec_eq_name()), major, zero);
7361         expr dec_eq_type = mk_bool();
7362         expr c = mk_simple_decl(dec_eq, dec_eq_type);
7363         expr minor_z = args[2];
7364         minor_z = visit_minor(minor_z);
7365         expr minor_s = args[3];
7366         expr pred = mk_app(mk_constant(get_nat_sub_name()), major, one);
7367         minor_s =
7368             ::lean::mk_let(next_name(), nat_type, pred, binding_body(minor_s));
7369         minor_s = visit_minor(minor_s);
7370         return mk_app(mk_constant(get_bool_cases_on_name()), c, minor_s,
7371                       minor_z);
7372     }
7373
7374     expr elim_int_cases(buffer<expr> &args) {
7375         lean_assert(args.size() == 4);
7376         expr major = visit(args[1]);
7377         expr zero = mk_lit(literal(nat(0)));
7378         expr int_type = mk_constant(get_int_name());
7379         expr nat_type = mk_constant(get_nat_name());
7380         expr izero = mk_simple_decl(
7381             mk_app(mk_constant(get_int_of_nat_name()), zero), int_type);
7382         expr dec_lt = mk_app(mk_constant(get_int_dec_lt_name()), major, izero);
7383         expr dec_lt_type = mk_bool();
7384         expr c = mk_simple_decl(dec_lt, dec_lt_type);
7385         expr abs = mk_app(mk_constant(get_int_nat_abs_name()), major);
7386         expr minor_p = args[2];
7387         minor_p =
7388             ::lean::mk_let(next_name(), nat_type, abs, binding_body(minor_p));
7389         minor_p = visit_minor(minor_p);
7390         expr one = mk_lit(literal(nat(1)));
7391         expr minor_n = args[3];
7392         minor_n = ::lean::mk_let(
7393             next_name(), nat_type, abs,
7394             ::lean::mk_let(
7395                 next_name(), nat_type,
7396                 mk_app(mk_constant(get_nat_sub_name()), mk_bvar(0), one),
7397                 binding_body(minor_n)));
7398         minor_n = visit_minor(minor_n);
7399         return mk_app(mk_constant(get_bool_cases_on_name()), c, minor_p,
7400                       minor_n);
7401     }
7402
7403     expr elim_array_cases(buffer<expr> &args) {
7404         lean_assert(args.size() == 4);
7405         expr major = visit(args[2]);
7406         expr minor = visit_minor(args[3]);
7407         lean_assert(is_lambda(minor));
7408         return ::lean::mk_let(
7409             next_name(), mk_enf_object_type(),
7410             mk_app(mk_constant(get_array_data_name()), mk_enf_neutral(), major),
7411             binding_body(minor));
7412     }
7413
7414     expr decidable_to_bool_cases(buffer<expr> const &args) {
7415         lean_assert(args.size() == 5);
7416         expr const &major = args[2];
7417         expr minor1 = args[3];
7418         expr minor2 = args[4];
7419         minor1 = visit_minor(minor1);
```

```
7420         minor2 = visit_minor(minor2);
7421         lean_assert(is_lambda(minor1));
7422         lean_assert(is_lambda(minor2));
7423         minor1 = instantiate(binding_body(minor1), mk_enf_neutral());
7424         minor2 = instantiate(binding_body(minor2), mk_enf_neutral());
7425         return mk_app(mk_constant(get_bool_cases_on_name()), major, minor1,
7426                       minor2);
7427     }
7428
7429     /* Remark: we only keep major and minor premises. */
7430     expr visit_cases_on(expr const &c, buffer<expr> &args) {
7431         name const &I_name = const_name(c).get_prefix();
7432         if (I_name == get_string_name()) {
7433             return elim_string_cases(args);
7434         } else if (I_name == get_nat_name()) {
7435             return elim_nat_cases(args);
7436         } else if (I_name == get_int_name()) {
7437             return elim_int_cases(args);
7438         } else if (I_name == get_array_name()) {
7439             return elim_array_cases(args);
7440         } else if (I_name == get_decidable_name()) {
7441             return decidable_to_bool_cases(args);
7442         } else {
7443             unsigned minors_begin;
7444             unsigned minors_end;
7445             std::tie(minors_begin, minors_end) =
7446                 get_cases_on_minors_range(env(), const_name(c));
7447             if (optional<unsigned> fidx =
7448                     has_trivial_structure(const_name(c).get_prefix())) {
7449                 /* Eliminate `cases_on` of trivial structure */
7450                 lean_assert(minors_end == minors_begin + 1);
7451                 expr major = args[minors_begin - 1];
7452                 lean_assert(is_atom(major));
7453                 expr minor = args[minors_begin];
7454                 unsigned i = 0;
7455                 buffer<expr> fields;
7456                 while (is_lambda(minor)) {
7457                     expr v = mk_proj(I_name, i, major);
7458                     expr t = infer_type(v);
7459                     name n = next_name();
7460                     expr fvar = m_lctx.mk_local_decl(ngen(), n, t, v);
7461                     fields.push_back(fvar);
7462                     expr new_t;
7463                     expr new_v;
7464                     if (*fidx == i) {
7465                         expr major_type = infer_type(major);
7466                         new_t = mk_runtime_type(major_type);
7467                         new_v = visit(major);
7468                     } else {
7469                         new_t = mk_enf_object_type();
7470                         new_v = mk_enf_neutral();
7471                     }
7472                     m_let_fvars.push_back(fvar);
7473                     m_let_entries.emplace_back(n, new_t, new_v);
7474                     i++;
7475                     minor = binding_body(minor);
7476                 }
7477                 expr r = instantiate_rev(minor, fields.size(), fields.data());
7478                 return visit(r);
7479             } else {
7480                 buffer<expr> new_args;
7481                 new_args.push_back(visit(args[minors_begin - 1]));
7482                 for (unsigned i = minors_begin; i < minors_end; i++) {
7483                     new_args.push_back(visit_minor(args[i]));
7484                 }
7485                 return mk_app(c, new_args);
7486             }
7487         }
7488     }
7489
```

```
7490     expr visit_app_default(expr fn, buffer<expr> &args) {
7491         fn = visit(fn);
7492         for (expr &arg : args) {
7493             if (!is_atom(arg)) {
7494                 // In LCNF, relevant arguments are atomic
7495                 arg = mk_enf_neutral();
7496             } else {
7497                 arg = visit(arg);
7498             }
7499         }
7500         return mk_app(fn, args);
7501     }
7502
7503     expr visit_quot_lift(buffer<expr> &args) {
7504         lean_assert(args.size() >= 6);
7505         expr f = args[3];
7506         buffer<expr> new_args;
7507         for (unsigned i = 5; i < args.size(); i++) new_args.push_back(args[i]);
7508         return visit_app_default(f, new_args);
7509     }
7510
7511     expr visit_quot_mk(buffer<expr> const &args) {
7512         lean_assert(args.size() == 3);
7513         return visit(args[2]);
7514     }
7515
7516     expr visit_constructor(expr const &fn, buffer<expr> &args) {
7517         constructor_val c_val = env().get(const_name(fn)).to_constructor_val();
7518         name const &I_name = c_val.get_induct();
7519         if (optional<unsigned> fidx = has_trivial_structure(I_name)) {
7520             unsigned nparams = c_val.get_nparams();
7521             lean_assert(nparams + *fidx < args.size());
7522             return visit(args[nparams + *fidx]);
7523         } else {
7524             return visit_app_default(fn, args);
7525         }
7526     }
7527
7528     expr visit_app(expr const &e) {
7529         buffer<expr> args;
7530         expr f = get_app_args(e, args);
7531         if (is_constant(f)) {
7532             name const &fn = const_name(f);
7533             if (fn == get_lc_proof_name()) {
7534                 return mk_enf_neutral();
7535             } else if (fn == get_lc_unreachable_name()) {
7536                 return mk_enf_unreachable();
7537             } else if (fn == get_decidable_is_true_name()) {
7538                 return mk_constant(get_bool_true_name());
7539             } else if (fn == get_decidable_is_false_name()) {
7540                 return mk_constant(get_bool_false_name());
7541             } else if (is_constructor(env(), fn)) {
7542                 return visit_constructor(f, args);
7543             } else if (is_cases_on_recursor(env(), fn)) {
7544                 return visit_cases_on(f, args);
7545             } else if (fn == get_quot_mk_name()) {
7546                 return visit_quot_mk(args);
7547             } else if (fn == get_quot_lift_name()) {
7548                 return visit_quot_lift(args);
7549             } else if (fn == get_decidable_decide_name() && args.size() == 2) {
7550                 /* Decidable.decide is the "identify" function since Decidable
7551                    and Bool have the same runtime representation. */
7552                 return args[1];
7553             }
7554         }
7555         return visit_app_default(f, args);
7556     }
7557
7558     expr visit_proj(expr const &e) {
7559         if (optional<unsigned> fidx = has_trivial_structure(proj_sname(e))) {
```

```
7560            if (*fidx != proj_idx(e).get_small_value())
7561                return mk_enf_neutral();
7562            else
7563                return visit(proj_expr(e));
7564        } else {
7565            return update_proj(e, visit(proj_expr(e)));
7566        }
7567    }
7568
7569    expr mk_let(unsigned saved_fvars_size, expr r) {
7570        lean_assert(saved_fvars_size <= m_let_fvars.size());
7571        lean_assert(m_let_fvars.size() == m_let_entries.size());
7572        if (saved_fvars_size == m_let_fvars.size()) return r;
7573        r = abstract(r, m_let_fvars.size() - saved_fvars_size,
7574                     m_let_fvars.data() + saved_fvars_size);
7575        unsigned i = m_let_fvars.size();
7576        while (i > saved_fvars_size) {
7577            --i;
7578            expr v =
7579                abstract(std::get<2>(m_let_entries[i]), i - saved_fvars_size,
7580                         m_let_fvars.data() + saved_fvars_size);
7581            r = ::lean::mk_let(std::get<0>(m_let_entries[i]),
7582                               std::get<1>(m_let_entries[i]), v, r);
7583        }
7584        m_let_fvars.shrink(saved_fvars_size);
7585        m_let_entries.shrink(saved_fvars_size);
7586        return r;
7587    }
7588
7589    expr visit_let(expr e) {
7590        lean_assert(m_let_entries.size() == m_let_fvars.size());
7591        buffer<expr> curr_fvars;
7592        while (is_let(e)) {
7593            expr t = instantiate_rev(let_type(e), curr_fvars.size(),
7594                                     curr_fvars.data());
7595            expr v = instantiate_rev(let_value(e), curr_fvars.size(),
7596                                     curr_fvars.data());
7597            name n = let_name(e);
7598            /* Pseudo "do" joinpoints are used to implement a temporary HACK.
7599             * See `visit_let` method at `lcnf.cpp` */
7600            if (is_internal_name(n) && !is_join_point_name(n) &&
7601                !is_pseudo_do_join_point_name(n)) {
7602                n = next_name();
7603            }
7604            expr fvar = m_lctx.mk_local_decl(ngen(), n, t, v);
7605            curr_fvars.push_back(fvar);
7606            expr new_t = mk_runtime_type(t);
7607            expr new_v = visit(v);
7608            m_let_fvars.push_back(fvar);
7609            m_let_entries.emplace_back(n, new_t, new_v);
7610            e = let_body(e);
7611        }
7612        lean_assert(m_let_entries.size() == m_let_fvars.size());
7613        return visit(instantiate_rev(e, curr_fvars.size(), curr_fvars.data()));
7614    }
7615
7616    expr visit_mdata(expr const &e) {
7617        return update_mdata(e, visit(mdata_expr(e)));
7618    }
7619
7620    expr visit(expr const &e) {
7621        lean_assert(m_let_entries.size() == m_let_fvars.size());
7622        switch (e.kind()) {
7623            case expr_kind::BVar:
7624            case expr_kind::MVar:
7625                lean_unreachable();
7626            case expr_kind::FVar:
7627                return visit_fvar(e);
7628            case expr_kind::Sort:
7629                return mk_enf_neutral();
```

```
            case expr_kind::Lit:
                return e;
            case expr_kind::Pi:
                return mk_enf_neutral();
            case expr_kind::Const:
                return visit_constant(e);
            case expr_kind::App:
                return visit_app(e);
            case expr_kind::Proj:
                return visit_proj(e);
            case expr_kind::MData:
                return visit_mdata(e);
            case expr_kind::Lambda:
                return visit_lambda(e);
            case expr_kind::Let:
                return visit_let(e);
        }
        lean_unreachable();
    }

   public:
    erase_irrelevant_fn(environment const &env, local_ctx const &lctx)
        : m_st(env), m_lctx(lctx), m_x("_x") {}
    expr operator()(expr const &e) { return mk_let(0, visit(e)); }
};

expr erase_irrelevant_core(environment const &env, local_ctx const &lctx,
                           expr const &e) {
    return erase_irrelevant_fn(env, lctx)(e);
}
}  // namespace lean
// ::::::::::::::
// compiler/export_attribute.cpp
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include "library/constants.h"
#include "library/util.h"

namespace lean {
extern "C" object *lean_get_export_name_for(object *env, object *n);
optional<name> get_export_name_for(environment const &env, name const &n) {
    return to_optional<name>(
        lean_get_export_name_for(env.to_obj_arg(), n.to_obj_arg()));
}
}  // namespace lean
// ::::::::::::::
// compiler/extern_attribute.cpp
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include <string>

#include "kernel/instantiate.h"
#include "kernel/type_checker.h"
#include "library/compiler/borrowed_annotation.h"
#include "library/compiler/extern_attribute.h"
#include "library/compiler/ir.h"
#include "library/compiler/util.h"
#include "library/projection.h"
```

```
#include "library/util.h"
#include "util/io.h"
#include "util/object_ref.h"
#include "util/option_ref.h"

namespace lean {
extern "C" object *lean_get_extern_attr_data(object *env, object *n);

optional<extern_attr_data_value> get_extern_attr_data(environment const &env,
                                                      name const &fn) {
    return to_optional<extern_attr_data_value>(
        lean_get_extern_attr_data(env.to_obj_arg(), fn.to_obj_arg()));
}

bool is_extern_constant(environment const &env, name const &c) {
    return static_cast<bool>(get_extern_attr_data(env, c));
}

extern "C" object *lean_get_extern_const_arity(object *env, object *,
                                               object *w);

optional<unsigned> get_extern_constant_arity(environment const &env,
                                             name const &c) {
    auto arity = get_io_result<option_ref<nat>>(lean_get_extern_const_arity(
        env.to_obj_arg(), c.to_obj_arg(), lean_io_mk_world()));
    if (optional<nat> aux = arity.get()) {
        return optional<unsigned>(aux->get_small_value());
    } else {
        return optional<unsigned>();
    }
}

bool get_extern_borrowed_info(environment const &env, name const &c,
                              buffer<bool> &borrowed_args, bool &borrowed_res) {
    if (is_extern_constant(env, c)) {
        /* Extract borrowed info from type */
        expr type = env.get(c).get_type();
        unsigned arity = 0;
        while (is_pi(type)) {
            arity++;
            expr d = binding_domain(type);
            borrowed_args.push_back(is_borrowed(d));
            type = binding_body(type);
        }
        borrowed_res = false;
        if (optional<unsigned> c_arity = get_extern_constant_arity(env, c)) {
            if (*c_arity < arity) {
                borrowed_args.shrink(*c_arity);
                return true;
            } else if (*c_arity > arity) {
                borrowed_args.resize(*c_arity, false);
                return true;
            }
        }
        borrowed_res = is_borrowed(type);
        return true;
    }
    return false;
}

optional<expr> get_extern_constant_ll_type(environment const &env,
                                           name const &c) {
    if (is_extern_constant(env, c)) {
        unsigned arity = 0;
        expr type = env.get(c).get_type();
        type_checker::state st(env);
        local_ctx lctx;
        name_generator ngen;
        buffer<expr> arg_ll_types;
        buffer<expr> locals;
```

```
7770         while (is_pi(type)) {
7771             arity++;
7772             expr arg_type = instantiate_rev(binding_domain(type), locals.size(),
7773                                             locals.data());
7774             expr arg_ll_type = mk_runtime_type(st, lctx, arg_type);
7775             arg_ll_types.push_back(arg_ll_type);
7776             expr local = lctx.mk_local_decl(ngen, binding_name(type), arg_type);
7777             locals.push_back(local);
7778             type = binding_body(type);
7779         }
7780         type = instantiate_rev(type, locals.size(), locals.data());
7781         expr ll_type;
7782         if (optional<unsigned> c_arity = get_extern_constant_arity(env, c)) {
7783             if (arity < *c_arity) {
7784                 /* Fill with `_obj` */
7785                 arg_ll_types.resize(*c_arity, mk_enf_object_type());
7786                 ll_type = mk_enf_object_type();
7787             } else if (arity > *c_arity) {
7788                 arg_ll_types.shrink(*c_arity);
7789                 ll_type = mk_enf_object_type(); /* Result is a closure */
7790             } else {
7791                 ll_type = mk_runtime_type(st, lctx, type);
7792             }
7793         } else {
7794             ll_type = mk_runtime_type(st, lctx, type);
7795         }
7796         unsigned i = arg_ll_types.size();
7797         while (i > 0) {
7798             --i;
7799             ll_type = mk_arrow(arg_ll_types[i], ll_type);
7800         }
7801         return some_expr(ll_type);
7802     }
7803     return none_expr();
7804 }
7805 }  // namespace lean
7806 // ::::::::::::::
7807 // compiler/extract_closed.cpp
7808 // ::::::::::::::
7809 /*
7810 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
7811 Released under Apache 2.0 license as described in the file LICENSE.
7812
7813 Author: Leonardo de Moura
7814 */
7815 #include <lean/flet.h>
7816
7817 #include "kernel/expr_maps.h"
7818 #include "kernel/for_each_fn.h"
7819 #include "kernel/instantiate.h"
7820 #include "library/compiler/closed_term_cache.h"
7821 #include "library/compiler/reduce_arity.h"
7822 #include "library/compiler/util.h"
7823 #include "library/trace.h"
7824
7825 namespace lean {
7826 name mk_extract_closed_aux_fn(name const &n, unsigned idx) {
7827     return name(n, "_closed").append_after(idx);
7828 }
7829
7830 bool is_extract_closed_aux_fn(name const &n) {
7831     if (!n.is_string() || n.is_atomic()) return false;
7832     return strncmp(n.get_string().data(), "_closed", 7) == 0;
7833 }
7834
7835 class extract_closed_fn {
7836     environment m_env;
7837     name_generator m_ngen;
7838     local_ctx m_lctx;
7839     buffer<comp_decl> m_new_decls;
```

```
7840     name m_base_name;
7841     unsigned m_next_idx{1};
7842     expr_map<bool> m_closed;
7843
7844     environment const &env() const { return m_env; }
7845     name_generator &ngen() { return m_ngen; }
7846
7847     name next_name() {
7848         name r = mk_extract_closed_aux_fn(m_base_name, m_next_idx);
7849         m_next_idx++;
7850         return r;
7851     }
7852
7853     expr find(expr const &e) {
7854         if (is_fvar(e)) {
7855             if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
7856                 if (optional<expr> v = decl->get_value()) {
7857                     return find(*v);
7858                 }
7859             }
7860         } else if (is_mdata(e)) {
7861             return find(mdata_expr(e));
7862         }
7863         return e;
7864     }
7865
7866     bool is_closed(expr e) {
7867         switch (e.kind()) {
7868             case expr_kind::MVar:
7869                 lean_unreachable();
7870             case expr_kind::Pi:
7871                 lean_unreachable();
7872             case expr_kind::Sort:
7873                 lean_unreachable();
7874             case expr_kind::Lit:
7875                 return true;
7876             case expr_kind::BVar:
7877                 return true;
7878             case expr_kind::Const:
7879                 return true;
7880             case expr_kind::MData:
7881                 return is_closed(mdata_expr(e));
7882             case expr_kind::Proj:
7883                 return is_closed(proj_expr(e));
7884             default:
7885                 break;
7886         };
7887
7888         auto it = m_closed.find(e);
7889         if (it != m_closed.end()) return it->second;
7890
7891         bool r;
7892         switch (e.kind()) {
7893             case expr_kind::FVar:
7894                 if (auto v = m_lctx.get_local_decl(e).get_value()) {
7895                     r = is_closed(*v);
7896                 } else {
7897                     r = false;
7898                 }
7899                 break;
7900             case expr_kind::App: {
7901                 buffer<expr> args;
7902                 expr const &fn = get_app_args(e, args);
7903                 r = true;
7904                 if (!is_closed(fn)) {
7905                     r = false;
7906                 } else {
7907                     if (is_constant(fn) &&
7908                         has_never_extract_attribute(m_env, const_name(fn))) {
7909                         r = false;
```

```
7910                    } else {
7911                        for (expr const &arg : args) {
7912                            if (!is_closed(arg)) {
7913                                r = false;
7914                                break;
7915                            }
7916                        }
7917                    }
7918                }
7919                break;
7920            }
7921            case expr_kind::Lambda:
7922                while (is_lambda(e)) {
7923                    e = binding_body(e);
7924                }
7925                r = is_closed(e);
7926                break;
7927            case expr_kind::Let:
7928                r = true;
7929                while (is_let(e)) {
7930                    if (!is_closed(let_value(e))) {
7931                        r = false;
7932                        break;
7933                    }
7934                    e = let_body(e);
7935                }
7936                if (r && !is_closed(e)) {
7937                    r = false;
7938                }
7939                break;
7940            default:
7941                lean_unreachable();
7942        }
7943        m_closed.insert(mk_pair(e, r));
7944        return r;
7945    }
7946
7947    expr visit_lambda(expr e) {
7948        flet<local_ctx> save_lctx(m_lctx, m_lctx);
7949        buffer<expr> fvars;
7950        while (is_lambda(e)) {
7951            lean_assert(!has_loose_bvars(binding_domain(e)));
7952            expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
7953                                                 binding_domain(e));
7954            fvars.push_back(new_fvar);
7955            e = binding_body(e);
7956        }
7957        expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
7958        return m_lctx.mk_lambda(fvars, r);
7959    }
7960
7961    bool is_neutral_constructor_app(expr const &e) {
7962        if (!is_constructor_app(env(), e)) return false;
7963        buffer<expr> args;
7964        get_app_args(e, args);
7965        for (expr const &arg : args) {
7966            if (!is_enf_neutral(arg)) return false;
7967        }
7968        return true;
7969    }
7970
7971    void collect_deps(expr e, name_set &collected, buffer<expr> &fvars) {
7972        buffer<expr> todo;
7973        while (true) {
7974            for_each(e, [&](expr const &x, unsigned) {
7975                if (!has_fvar(x)) return false;
7976                if (is_fvar(x) && !collected.contains(fvar_name(x))) {
7977                    collected.insert(fvar_name(x));
7978                    optional<expr> v = m_lctx.get_local_decl(x).get_value();
7979                    lean_assert(v);
```

```
                    fvars.push_back(x);
                    todo.push_back(*v);
                }
                return true;
            });
            if (todo.empty()) return;
            e = todo.back();
            todo.pop_back();
        }
    }

    void collect_deps(expr e, buffer<expr> &fvars) {
        name_set collected;
        collect_deps(e, collected, fvars);
        sort_fvars(m_lctx, fvars);
    }

    bool arity_eq_0(name c) {
        c = mk_cstage2_name(c);
        optional<constant_info> info = env().find(c);
        if (!info || !info->is_definition()) return false;
        return !is_lambda(info->get_value());
    }

    bool is_join_point_app(expr const &e) const {
        if (!is_app(e)) return false;
        expr const &fn = get_app_fn(e);
        return is_fvar(fn) &&
               is_join_point_name(m_lctx.get_local_decl(fn).get_user_name());
    }

    expr mk_aux_constant(expr const &e0) {
        expr e = find(e0);
        if (is_enf_neutral(e) || is_enf_unreachable(e)) {
            return e0;
        }
        if (is_join_point_app(e)) {
            return e0;
        }
        if (is_constant(e) && arity_eq_0(const_name(e))) {
            /* Remarr: if a constant `C` has arity > 0, then it is worth
               creating a new constant with arity 0 that just returns `C`. In
               this way, we cache the closure allocation. To implement this
               optimization we need to first store the definitions after
               erasure. */
            return e0;
        }
        if (is_neutral_constructor_app(e)) {
            /* We don't create auxiliary constants for constructor applications
               such as: `none ■` and `list.nil ■` */
            return e0;
        }
        if (is_lit(e) && lit_value(e).kind() == literal_kind::Nat &&
            lit_value(e).get_nat().is_small()) {
            /* We don't create auxiliary constants for small nat literals.
             * Reason: they are cheap. */
            return e0;
        }
        if (!is_lit(e) && is_morally_num_lit(e)) {
            /* We don't create auxiliary constants for uint* literals. */
            return e0;
        }
        buffer<expr> fvars;
        collect_deps(e, fvars);
        e = m_lctx.mk_lambda(fvars, e);
        lean_assert(!has_loose_bvars(e));
        if (optional<name> c = get_closed_term_name(m_env, e)) {
            return mk_constant(*c);
        }
        name c = next_name();
```

```
8050         m_new_decls.push_back(comp_decl(c, e));
8051         m_env = cache_closed_term_name(m_env, e, c);
8052         return mk_constant(c);
8053     }
8054
8055     expr visit_let(expr e) {
8056         flet<local_ctx> save_lctx(m_lctx, m_lctx);
8057         buffer<expr> fvars;
8058         while (is_let(e)) {
8059             lean_assert(!has_loose_bvars(let_type(e)));
8060             expr new_val = visit(
8061                 instantiate_rev(let_value(e), fvars.size(), fvars.data()));
8062             expr new_fvar =
8063                 m_lctx.mk_local_decl(ngen(), let_name(e), let_type(e), new_val);
8064             fvars.push_back(new_fvar);
8065             e = let_body(e);
8066         }
8067         expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
8068         return m_lctx.mk_lambda(fvars, r);
8069     }
8070
8071     expr visit_app(expr const &e) {
8072         buffer<expr> args;
8073         expr const &fn = get_app_args(e, args);
8074         for (unsigned i = 0; i < args.size(); i++) {
8075             args[i] = visit(args[i]);
8076         }
8077         expr r = mk_app(fn, args);
8078         if (is_closed(r))
8079             return mk_aux_constant(r);
8080         else
8081             return r;
8082     }
8083
8084     expr visit_atom(expr const &e) { return mk_aux_constant(e); }
8085
8086     expr visit(expr const &e) {
8087         switch (e.kind()) {
8088             case expr_kind::Lit:
8089                 return visit_atom(e);
8090             case expr_kind::Const:
8091                 return visit_atom(e);
8092             case expr_kind::App:
8093                 return visit_app(e);
8094             case expr_kind::Lambda:
8095                 return visit_lambda(e);
8096             case expr_kind::Let:
8097                 return visit_let(e);
8098             default:
8099                 return e;
8100         }
8101     }
8102
8103    public:
8104     extract_closed_fn(environment const &env) : m_env(env) {}
8105
8106     pair<environment, comp_decls> operator()(comp_decl const &d) {
8107         if (arity_was_reduced(d)) {
8108             /* Do nothing since `d` will be inlined. */
8109             return mk_pair(env(), comp_decls(d));
8110         }
8111         expr v = d.snd();
8112         if (is_extract_closed_aux_fn(d.fst())) {
8113             /* Do not extract closed terms from an auxiliary declaration created
8114              * by this module. */
8115             return mk_pair(env(), comp_decls(d));
8116         }
8117         m_base_name = d.fst();
8118         expr new_v = visit(v);
8119         comp_decl new_d(d.fst(), new_v);
```

```
8120         m_new_decls.push_back(new_d);
8121         return mk_pair(env(), comp_decls(m_new_decls));
8122     }
8123 };
8124
8125 pair<environment, comp_decls> extract_closed_core(environment const &env,
8126                                                    comp_decl const &d) {
8127     return extract_closed_fn(env)(d);
8128 }
8129
8130 pair<environment, comp_decls> extract_closed(environment env,
8131                                              comp_decls const &ds) {
8132     comp_decls r;
8133     for (comp_decl const &d : ds) {
8134         comp_decls new_ds;
8135         std::tie(env, new_ds) = extract_closed_core(env, d);
8136         r = append(r, new_ds);
8137     }
8138     return mk_pair(env, r);
8139 }
8140 }  // namespace lean
8141 // ::::::::::::::
8142 // compiler/find_jp.cpp
8143 // ::::::::::::::
8144 /*
8145 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
8146 Released under Apache 2.0 license as described in the file LICENSE.
8147
8148 Author: Leonardo de Moura
8149 */
8150 #include "kernel/abstract.h"
8151 #include "kernel/for_each_fn.h"
8152 #include "kernel/instantiate.h"
8153 #include "library/compiler/util.h"
8154
8155 namespace lean {
8156
8157 /* Find join-points */
8158 class find_jp_fn {
8159     environment const &m_env;
8160     local_ctx m_lctx;
8161     name_generator m_ngen;
8162     name_map<unsigned> m_candidates;
8163
8164     /* Remove all candidates occurring in `e`. */
8165     void remove_candidates_occurring_at(expr const &e) {
8166         for_each(e, [&](expr const &e, unsigned) {
8167             if (!has_fvar(e)) return false;
8168             if (is_fvar(e)) {
8169                 m_candidates.erase(fvar_name(e));
8170             }
8171             return true;
8172         });
8173     }
8174
8175     expr visit_lambda(expr e) {
8176         buffer<expr> fvars;
8177         while (is_lambda(e)) {
8178             expr domain =
8179                 instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
8180             remove_candidates_occurring_at(domain);
8181             expr fvar = m_lctx.mk_local_decl(m_ngen, binding_name(e), domain,
8182                                              binding_info(e));
8183             fvars.push_back(fvar);
8184             e = binding_body(e);
8185         }
8186         e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
8187         return m_lctx.mk_lambda(fvars, e);
8188     }
8189
```

```
8190     expr visit_cases(expr const &e) {
8191         lean_assert(is_cases_on_app(m_env, e));
8192         buffer<expr> args;
8193         expr const &c = get_app_args(e, args);
8194         /* simplify minor premises */
8195         unsigned minor_idx;
8196         unsigned minors_end;
8197         bool before_erasure = true;
8198         std::tie(minor_idx, minors_end) =
8199             get_cases_on_minors_range(m_env, const_name(c), before_erasure);
8200         for (unsigned i = 0; i < minor_idx; i++) {
8201             remove_candidates_occurring_at(args[i]);
8202         }
8203         for (; minor_idx < minors_end; minor_idx++) {
8204             args[minor_idx] = visit(args[minor_idx]);
8205         }
8206         for (unsigned i = minors_end; i < args.size(); i++) {
8207             remove_candidates_occurring_at(args[i]);
8208         }
8209         return mk_app(c, args);
8210     }
8211
8212     expr visit_app(expr const &e) {
8213         lean_assert(is_app(e));
8214         if (is_cases_on_app(m_env, e)) {
8215             return visit_cases(e);
8216         } else {
8217             buffer<expr> args;
8218             expr const &fn = get_app_args(e, args);
8219             for (expr const &arg : args) remove_candidates_occurring_at(arg);
8220             if (is_fvar(fn)) {
8221                 if (unsigned const *arity = m_candidates.find(fvar_name(fn))) {
8222                     if (args.size() != *arity)
8223                         remove_candidates_occurring_at(fn);
8224                 }
8225             }
8226             return e;
8227         }
8228     }
8229
8230     expr visit_let(expr e) {
8231         buffer<expr> fvars;
8232         while (is_let(e)) {
8233             expr new_type =
8234                 instantiate_rev(let_type(e), fvars.size(), fvars.data());
8235             remove_candidates_occurring_at(new_type);
8236             expr new_val =
8237                 instantiate_rev(let_value(e), fvars.size(), fvars.data());
8238             expr fvar =
8239                 m_lctx.mk_local_decl(m_ngen, let_name(e), new_type, new_val);
8240             fvars.push_back(fvar);
8241             if (is_lambda(new_val)) {
8242                 unsigned arity = get_num_nested_lambdas(new_val);
8243                 m_candidates.insert(fvar_name(fvar), arity);
8244             }
8245             e = let_body(e);
8246         }
8247         e = instantiate_rev(e, fvars.size(), fvars.data());
8248         e = visit(e);
8249         e = abstract(e, fvars.size(), fvars.data());
8250         unsigned i = fvars.size();
8251         while (i > 0) {
8252             --i;
8253             expr const &fvar = fvars[i];
8254             local_decl fvar_decl = m_lctx.get_local_decl(fvar);
8255             expr type = fvar_decl.get_type();
8256             expr value = *fvar_decl.get_value();
8257             name n = fvar_decl.get_user_name();
8258             if (m_candidates.contains(fvar_name(fvar))) {
8259                 value = visit(value);
```

```
8260                 n = mk_join_point_name(n);
8261             } else {
8262                 remove_candidates_occurring_at(value);
8263             }
8264             type = abstract(type, i, fvars.data());
8265             value = abstract(value, i, fvars.data());
8266             e = mk_let(n, type, value, e);
8267         }
8268         return e;
8269     }
8270
8271     expr visit(expr const &e) {
8272         switch (e.kind()) {
8273             case expr_kind::Lambda:
8274                 return visit_lambda(e);
8275             case expr_kind::Let:
8276                 return visit_let(e);
8277             case expr_kind::App:
8278                 return visit_app(e);
8279             case expr_kind::MData:
8280                 return update_mdata(e, visit(mdata_expr(e)));
8281             default:
8282                 return e;
8283         }
8284     }
8285
8286    public:
8287     find_jp_fn(environment const &env) : m_env(env) {}
8288
8289     expr operator()(expr const &e) { return visit(e); }
8290 };
8291
8292 expr find_jp(environment const &env, expr const &e) {
8293     return find_jp_fn(env)(e);
8294 }
8295 }  // namespace lean
8296 //::::::::::::::
8297 // compiler/implemented_by_attribute.cpp
8298 //::::::::::::::
8299 /*
8300 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
8301 Released under Apache 2.0 license as described in the file LICENSE.
8302
8303 Author: Leonardo de Moura
8304 */
8305 #include "kernel/environment.h"
8306 namespace lean {
8307 extern "C" object *lean_get_implemented_by(object *, object *);
8308
8309 optional<name> get_implemented_by_attribute(environment const &env,
8310                                              name const &n) {
8311     return to_optional<name>(
8312         lean_get_implemented_by(env.to_obj_arg(), n.to_obj_arg()));
8313 }
8314 }  // namespace lean
8315 // ::::::::::::::
8316 // compiler/init_attribute.cpp
8317 // ::::::::::::::
8318 /*
8319 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
8320 Released under Apache 2.0 license as described in the file LICENSE.
8321
8322 Authors: Leonardo de Moura
8323 */
8324 #include "kernel/environment.h"
8325 #include "util/object_ref.h"
8326
8327 namespace lean {
8328 extern "C" object *lean_get_init_fn_name_for(object *env, object *fn);
8329
```

```
8330 optional<name> get_init_fn_name_for(environment const &env, name const &n) {
8331     return to_optional<name>(
8332         lean_get_init_fn_name_for(env.to_obj_arg(), n.to_obj_arg()));
8333 }
8334 }  // namespace lean
8335 // ::::::::::::::
8336 // compiler/init_module.cpp
8337 // ::::::::::::::
8338 /*
8339 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
8340 Released under Apache 2.0 license as described in the file LICENSE.
8341
8342 Author: Leonardo de Moura
8343 */
8344 #include "library/compiler/borrowed_annotation.h"
8345 #include "library/compiler/compiler.h"
8346 #include "library/compiler/cse.h"
8347 #include "library/compiler/elim_dead_let.h"
8348 #include "library/compiler/ir.h"
8349 #include "library/compiler/ir_interpreter.h"
8350 #include "library/compiler/lcnf.h"
8351 #include "library/compiler/ll_infer_type.h"
8352 #include "library/compiler/llnf.h"
8353 #include "library/compiler/specialize.h"
8354 #include "library/compiler/util.h"
8355
8356 namespace lean {
8357 void initialize_compiler_module() {
8358     initialize_compiler_util();
8359     initialize_lcnf();
8360     initialize_elim_dead_let();
8361     initialize_cse();
8362     initialize_specialize();
8363     initialize_llnf();
8364     initialize_compiler();
8365     initialize_borrowed_annotation();
8366     initialize_ll_infer_type();
8367     initialize_ir();
8368     initialize_ir_interpreter();
8369 }
8370
8371 void finalize_compiler_module() {
8372     finalize_ir_interpreter();
8373     finalize_ir();
8374     finalize_ll_infer_type();
8375     finalize_borrowed_annotation();
8376     finalize_compiler();
8377     finalize_llnf();
8378     finalize_specialize();
8379     finalize_cse();
8380     finalize_elim_dead_let();
8381     finalize_lcnf();
8382     finalize_compiler_util();
8383 }
8384 }  // namespace lean
8385 // ::::::::::::::
8386 // compiler/ir.cpp
8387 // ::::::::::::::
8388 /*
8389 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
8390 Released under Apache 2.0 license as described in the file LICENSE.
8391
8392 Author: Leonardo de Moura
8393 */
8394 #include <string>
8395
8396 #include "kernel/instantiate.h"
8397 #include "kernel/type_checker.h"
8398 #include "library/compiler/extern_attribute.h"
8399 #include "library/compiler/ir.h"
```

```
8400 #include "library/compiler/llnf.h"
8401 #include "library/compiler/util.h"
8402 #include "library/trace.h"
8403 #include "util/array_ref.h"
8404 #include "util/nat.h"
8405
8406 namespace lean {
8407 namespace ir {
8408 object *irrelevant_arg;
8409 extern "C" object *lean_ir_mk_unreachable(object *);
8410 extern "C" object *lean_ir_mk_var_arg(object *id);
8411 extern "C" object *lean_ir_mk_param(object *x, uint8 borrowed, object *ty);
8412 extern "C" object *lean_ir_mk_ctor_expr(object *n, object *cidx, object *size,
8413                                         object *usize, object *ssize,
8414                                         object *ys);
8415 extern "C" object *lean_ir_mk_proj_expr(object *i, object *x);
8416 extern "C" object *lean_ir_mk_uproj_expr(object *i, object *x);
8417 extern "C" object *lean_ir_mk_sproj_expr(object *n, object *o, object *x);
8418 extern "C" object *lean_ir_mk_fapp_expr(object *c, object *ys);
8419 extern "C" object *lean_ir_mk_papp_expr(object *c, object *ys);
8420 extern "C" object *lean_ir_mk_app_expr(object *x, object *ys);
8421 extern "C" object *lean_ir_mk_num_expr(object *v);
8422 extern "C" object *lean_ir_mk_str_expr(object *v);
8423 extern "C" object *lean_ir_mk_vdecl(object *x, object *ty, object *e,
8424                                     object *b);
8425 extern "C" object *lean_ir_mk_jdecl(object *j, object *xs, object *v,
8426                                     object *b);
8427 extern "C" object *lean_ir_mk_uset(object *x, object *i, object *y, object *b);
8428 extern "C" object *lean_ir_mk_sset(object *x, object *i, object *o, object *y,
8429                                    object *ty, object *b);
8430 extern "C" object *lean_ir_mk_case(object *tid, object *x, object *cs);
8431 extern "C" object *lean_ir_mk_ret(object *x);
8432 extern "C" object *lean_ir_mk_jmp(object *j, object *ys);
8433 extern "C" object *lean_ir_mk_alt(object *n, object *cidx, object *size,
8434                                   object *usize, object *ssize, object *b);
8435 extern "C" object *lean_ir_mk_decl(object *f, object *xs, object *ty,
8436                                    object *b);
8437 extern "C" object *lean_ir_mk_extern_decl(object *f, object *xs, object *ty,
8438                                           object *ext_entry);
8439 extern "C" object *lean_ir_decl_to_string(object *d);
8440 extern "C" object *lean_ir_compile(object *env, object *opts, object *decls);
8441 extern "C" object *lean_ir_log_to_string(object *log);
8442 extern "C" object *lean_ir_add_decl(object *env, object *decl);
8443
8444 arg mk_var_arg(var_id const &id) {
8445     inc(id.raw());
8446     return arg(lean_ir_mk_var_arg(id.raw()));
8447 }
8448 arg mk_irrelevant_arg() { return arg(irrelevant_arg); }
8449 object *box_type(type ty) { return box(static_cast<size_t>(ty)); }
8450 param mk_param(var_id const &x, type ty, bool borrowed = false) {
8451     return param(lean_ir_mk_param(x.to_obj_arg(), borrowed, box_type(ty)));
8452 }
8453 expr mk_ctor(name const &n, unsigned cidx, unsigned size, unsigned usize,
8454              unsigned ssize, buffer<arg> const &ys) {
8455     return expr(lean_ir_mk_ctor_expr(n.to_obj_arg(), mk_nat_obj(cidx),
8456                                      mk_nat_obj(size), mk_nat_obj(usize),
8457                                      mk_nat_obj(ssize), to_array(ys)));
8458 }
8459 expr mk_proj(unsigned i, var_id const &x) {
8460     return expr(lean_ir_mk_proj_expr(mk_nat_obj(i), x.to_obj_arg()));
8461 }
8462 expr mk_uproj(unsigned i, var_id const &x) {
8463     return expr(lean_ir_mk_uproj_expr(mk_nat_obj(i), x.to_obj_arg()));
8464 }
8465 expr mk_sproj(unsigned i, unsigned o, var_id const &x) {
8466     return expr(
8467         lean_ir_mk_sproj_expr(mk_nat_obj(i), mk_nat_obj(o), x.to_obj_arg()));
8468 }
8469 expr mk_fapp(fun_id const &c, buffer<arg> const &ys) {
```

```
8470     return expr(lean_ir_mk_fapp_expr(c.to_obj_arg(), to_array(ys)));
8471 }
8472 expr mk_papp(fun_id const &c, buffer<arg> const &ys) {
8473     return expr(lean_ir_mk_papp_expr(c.to_obj_arg(), to_array(ys)));
8474 }
8475 expr mk_app(var_id const &x, buffer<arg> const &ys) {
8476     return expr(lean_ir_mk_app_expr(x.to_obj_arg(), to_array(ys)));
8477 }
8478 expr mk_num_lit(nat const &v) {
8479     return expr(lean_ir_mk_num_expr(v.to_obj_arg()));
8480 }
8481 expr mk_str_lit(string_ref const &v) {
8482     return expr(lean_ir_mk_str_expr(v.to_obj_arg()));
8483 }
8484
8485 fn_body mk_vdecl(var_id const &x, type ty, expr const &e, fn_body const &b) {
8486     return fn_body(lean_ir_mk_vdecl(x.to_obj_arg(), box_type(ty),
8487                                     e.to_obj_arg(), b.to_obj_arg()));
8488 }
8489 fn_body mk_jdecl(jp_id const &j, buffer<param> const &xs, expr const &v,
8490                  fn_body const &b) {
8491     return fn_body(lean_ir_mk_jdecl(j.to_obj_arg(), to_array(xs),
8492                                     v.to_obj_arg(), b.to_obj_arg()));
8493 }
8494 fn_body mk_uset(var_id const &x, unsigned i, var_id const &y,
8495                 fn_body const &b) {
8496     return fn_body(lean_ir_mk_uset(x.to_obj_arg(), mk_nat_obj(i),
8497                                    y.to_obj_arg(), b.to_obj_arg()));
8498 }
8499 fn_body mk_sset(var_id const &x, unsigned i, unsigned o, var_id const &y,
8500                 type ty, fn_body const &b) {
8501     return fn_body(lean_ir_mk_sset(x.to_obj_arg(), mk_nat_obj(i), mk_nat_obj(o),
8502                                    y.to_obj_arg(), box_type(ty),
8503                                    b.to_obj_arg()));
8504 }
8505 fn_body mk_ret(arg const &x) { return fn_body(lean_ir_mk_ret(x.to_obj_arg())); }
8506 fn_body mk_unreachable() { return fn_body(lean_ir_mk_unreachable(box(0))); }
8507 alt mk_alt(name const &n, unsigned cidx, unsigned size, unsigned usize,
8508            unsigned ssize, fn_body const &b) {
8509     return alt(lean_ir_mk_alt(n.to_obj_arg(), mk_nat_obj(cidx),
8510                               mk_nat_obj(size), mk_nat_obj(usize),
8511                               mk_nat_obj(ssize), b.to_obj_arg()));
8512 }
8513 fn_body mk_case(name const &tid, var_id const &x, buffer<alt> const &alts) {
8514     return fn_body(
8515         lean_ir_mk_case(tid.to_obj_arg(), x.to_obj_arg(), to_array(alts)));
8516 }
8517 fn_body mk_jmp(jp_id const &j, buffer<arg> const &ys) {
8518     return fn_body(lean_ir_mk_jmp(j.to_obj_arg(), to_array(ys)));
8519 }
8520 decl mk_decl(fun_id const &f, buffer<param> const &xs, type ty,
8521              fn_body const &b) {
8522     return decl(lean_ir_mk_decl(f.to_obj_arg(), to_array(xs), box_type(ty),
8523                                 b.to_obj_arg()));
8524 }
8525 decl mk_extern_decl(fun_id const &f, buffer<param> const &xs, type ty,
8526                     extern_attr_data_value const &v) {
8527     return decl(lean_ir_mk_extern_decl(f.to_obj_arg(), to_array(xs),
8528                                        box_type(ty), v.to_obj_arg()));
8529 }
8530 std::string decl_to_string(decl const &d) {
8531     string_ref r(lean_ir_decl_to_string(d.to_obj_arg()));
8532     return r.to_std_string();
8533 }
8534 environment add_decl(environment const &env, decl const &d) {
8535     return environment(lean_ir_add_decl(env.to_obj_arg(), d.to_obj_arg()));
8536 }
8537 }  // namespace ir
8538
8539 static ir::type to_ir_type(expr const &e) {
```

```
8540     if (is_constant(e)) {
8541         if (e == mk_enf_object_type()) {
8542             return ir::type::Object;
8543         } else if (e == mk_enf_neutral_type()) {
8544             return ir::type::Irrelevant;
8545         } else if (const_name(e) == get_uint8_name()) {
8546             return ir::type::UInt8;
8547         } else if (const_name(e) == get_uint16_name()) {
8548             return ir::type::UInt16;
8549         } else if (const_name(e) == get_uint32_name()) {
8550             return ir::type::UInt32;
8551         } else if (const_name(e) == get_uint64_name()) {
8552             return ir::type::UInt64;
8553         } else if (const_name(e) == get_usize_name()) {
8554             return ir::type::USize;
8555         } else if (const_name(e) == get_float_name()) {
8556             return ir::type::Float;
8557         }
8558     } else if (is_pi(e)) {
8559         return ir::type::Object;
8560     }
8561     throw exception("IR unsupported type");
8562 }
8563
8564 class to_ir_fn {
8565     type_checker::state m_st;
8566     local_ctx m_lctx;
8567     name m_x{"x"};
8568     unsigned m_next_idx{1};
8569
8570     environment const &env() const { return m_st.env(); }
8571
8572     name_generator &ngen() { return m_st.ngen(); }
8573
8574     static bool is_jmp(expr const &e) { return is_llnf_jmp(get_app_fn(e)); }
8575
8576     name next_name() {
8577         name r(m_x, m_next_idx);
8578         m_next_idx++;
8579         return r;
8580     }
8581
8582     ir::var_id to_var_id(local_decl const &d) {
8583         name n = d.get_user_name();
8584         lean_assert(n.is_numeral());
8585         return n.get_numeral();
8586     }
8587
8588     ir::jp_id to_jp_id(local_decl const &d) { return to_var_id(d); }
8589
8590     ir::var_id to_var_id(expr const &e) {
8591         lean_assert(is_fvar(e));
8592         return to_var_id(m_lctx.get_local_decl(e));
8593     }
8594
8595     ir::jp_id to_jp_id(expr const &e) { return to_var_id(e); }
8596
8597     ir::arg to_ir_arg(expr const &e) {
8598         lean_assert(is_fvar(e) || is_enf_neutral(e));
8599         if (is_fvar(e))
8600             return ir::mk_var_arg(to_var_id(e));
8601         else
8602             return ir::mk_irrelevant_arg();
8603     }
8604
8605     ir::type to_ir_result_type(expr e, unsigned arity) {
8606         for (unsigned i = 0; i < arity; i++) {
8607             if (!is_pi(e)) return ir::type::Object;
8608             e = binding_body(e);
8609         }
```

```
8610        return to_ir_type(e);
8611    }
8612
8613    ir::type size_to_ir_type(unsigned sz) {
8614        switch (sz) {
8615            case 1:
8616                return ir::type::UInt8;
8617            case 2:
8618                return ir::type::UInt16;
8619            case 4:
8620                return ir::type::UInt32;
8621            case 8:
8622                return ir::type::UInt64;
8623            default:
8624                throw exception("unsupported type size");
8625        }
8626    }
8627
8628    ir::fn_body visit_lambda(expr e, buffer<ir::param> &new_xs) {
8629        buffer<expr> fvars;
8630        while (is_lambda(e)) {
8631            lean_assert(!has_loose_bvars(binding_domain(e)));
8632            expr new_fvar =
8633                m_lctx.mk_local_decl(ngen(), next_name(), binding_domain(e));
8634            new_xs.push_back(ir::mk_param(to_var_id(new_fvar),
8635                                          to_ir_type(binding_domain(e))));
8636            fvars.push_back(new_fvar);
8637            e = binding_body(e);
8638        }
8639        return to_ir_fn_body(instantiate_rev(e, fvars.size(), fvars.data()));
8640    }
8641
8642    void to_ir_args(unsigned sz, expr const *args, buffer<ir::arg> &result) {
8643        for (unsigned i = 0; i < sz; i++) {
8644            result.push_back(to_ir_arg(args[i]));
8645        }
8646    }
8647
8648    ir::fn_body visit_cases(expr const &e) {
8649        buffer<expr> args;
8650        expr const &c = get_app_args(e, args);
8651        lean_assert(is_constant(c));
8652        name const &I_name = const_name(c).get_prefix();
8653        buffer<name> cnames;
8654        get_constructor_names(env(), I_name, cnames);
8655        lean_assert(args.size() == cnames.size() + 1);
8656        ir::var_id x = to_var_id(args[0]);
8657        buffer<ir::alt> alts;
8658        for (unsigned i = 1; i < args.size(); i++) {
8659            cnstr_info cinfo = get_cnstr_info(m_st, cnames[i - 1]);
8660            ir::fn_body body = to_ir_fn_body(args[i]);
8661            alts.push_back(ir::mk_alt(cnames[i - 1], cinfo.m_cidx,
8662                                      cinfo.m_num_objs, cinfo.m_num_usizes,
8663                                      cinfo.m_scalar_sz, body));
8664        }
8665        return ir::mk_case(I_name, x, alts);
8666    }
8667
8668    ir::fn_body visit_jmp(expr const &e) {
8669        buffer<expr> args;
8670        get_app_args(e, args);
8671        expr const &jp = args[0];
8672        lean_assert(is_fvar(jp));
8673        buffer<ir::arg> ir_args;
8674        to_ir_args(args.size() - 1, args.data() + 1, ir_args);
8675        return ir::mk_jmp(to_jp_id(jp), ir_args);
8676    }
8677
8678    ir::fn_body visit_terminal(expr const &e) {
8679        if (is_cases_on_app(env(), e)) {
```

```
8680            return visit_cases(e);
8681        } else if (is_jmp(e)) {
8682            return visit_jmp(e);
8683        } else if (is_fvar(e) || is_enf_neutral(e)) {
8684            return ir::mk_ret(to_ir_arg(e));
8685        } else if (is_enf_unreachable(e)) {
8686            return ir::mk_unreachable();
8687        } else {
8688            lean_unreachable();
8689        }
8690    }
8691
8692    ir::expr visit_lit_val(expr const &val) {
8693        literal const &l = lit_value(val);
8694        switch (l.kind()) {
8695            case literal_kind::Nat:
8696                return ir::mk_num_lit(l.get_nat());
8697            case literal_kind::String:
8698                return ir::mk_str_lit(l.get_string());
8699        }
8700        lean_unreachable();
8701    }
8702
8703    ir::fn_body mk_vdecl(local_decl const &decl, ir::expr const &val,
8704                         ir::fn_body const &b) {
8705        ir::type type = to_ir_type(decl.get_type());
8706        return ir::mk_vdecl(to_var_id(decl), type, val, b);
8707    }
8708
8709    ir::fn_body visit_lit(local_decl const &decl, ir::fn_body const &b) {
8710        ir::expr val = visit_lit_val(*decl.get_value());
8711        return mk_vdecl(decl, val, b);
8712    }
8713
8714    ir::fn_body visit_jp(local_decl const &decl, ir::fn_body const &b) {
8715        expr val = *decl.get_value();
8716        buffer<ir::param> xs;
8717        ir::fn_body v = visit_lambda(val, xs);
8718        return ir::mk_jdecl(to_jp_id(decl), xs, v, b);
8719    }
8720
8721    ir::fn_body visit_ctor(local_decl const &decl, ir::fn_body const &b) {
8722        expr val = *decl.get_value();
8723        buffer<expr> args;
8724        expr const &fn = get_app_args(val, args);
8725        name I_name;
8726        unsigned cidx, num_usizes, num_bytes;
8727        lean_verify(is_llnf_cnstr(fn, I_name, cidx, num_usizes, num_bytes));
8728        buffer<name> cnames;
8729        get_constructor_names(env(), I_name, cnames);
8730        lean_assert(cidx < cnames.size());
8731        buffer<ir::arg> ir_args;
8732        to_ir_args(args.size(), args.data(), ir_args);
8733        ir::expr v = ir::mk_ctor(cnames[cidx], cidx, args.size(), num_usizes,
8734                                 num_bytes, ir_args);
8735        return mk_vdecl(decl, v, b);
8736    }
8737
8738    ir::fn_body visit_fapp(local_decl const &decl, ir::fn_body const &b) {
8739        expr val = *decl.get_value();
8740        buffer<expr> args;
8741        expr const &fn = get_app_args(val, args);
8742        lean_assert(is_constant(fn));
8743        buffer<ir::arg> ir_args;
8744        to_ir_args(args.size(), args.data(), ir_args);
8745        ir::expr v = ir::mk_fapp(const_name(fn), ir_args);
8746        return mk_vdecl(decl, v, b);
8747    }
8748
8749    ir::fn_body visit_papp(local_decl const &decl, ir::fn_body const &b) {
```

```
8750         expr val = *decl.get_value();
8751         buffer<expr> args;
8752         get_app_args(val, args);
8753         lean_assert(is_constant(args[0]));
8754         buffer<ir::arg> ir_args;
8755         to_ir_args(args.size() - 1, args.data() + 1, ir_args);
8756         ir::expr v = ir::mk_papp(const_name(args[0]), ir_args);
8757         return mk_vdecl(decl, v, b);
8758     }
8759
8760     ir::fn_body visit_app(local_decl const &decl, ir::fn_body const &b) {
8761         expr val = *decl.get_value();
8762         buffer<expr> args;
8763         get_app_args(val, args);
8764         buffer<ir::arg> ir_args;
8765         to_ir_args(args.size() - 1, args.data() + 1, ir_args);
8766         ir::expr v = ir::mk_app(to_var_id(args[0]), ir_args);
8767         return mk_vdecl(decl, v, b);
8768     }
8769
8770     ir::fn_body visit_sset(local_decl const &decl, ir::fn_body const &b) {
8771         expr val = *decl.get_value();
8772         buffer<expr> args;
8773         expr const &fn = get_app_args(val, args);
8774         lean_assert(args.size() == 2);
8775         unsigned sz, n, offset;
8776         lean_verify(is_llnf_sset(fn, sz, n, offset));
8777         return ir::mk_sset(to_var_id(args[0]), n, offset, to_var_id(args[1]),
8778                            size_to_ir_type(sz), b);
8779     }
8780
8781     ir::fn_body visit_fset(local_decl const &decl, ir::fn_body const &b) {
8782         expr val = *decl.get_value();
8783         buffer<expr> args;
8784         expr const &fn = get_app_args(val, args);
8785         lean_assert(args.size() == 2);
8786         unsigned n, offset;
8787         lean_verify(is_llnf_fset(fn, n, offset));
8788         return ir::mk_sset(to_var_id(args[0]), n, offset, to_var_id(args[1]),
8789                            ir::type::Float, b);
8790     }
8791
8792     ir::fn_body visit_uset(local_decl const &decl, ir::fn_body const &b) {
8793         expr val = *decl.get_value();
8794         buffer<expr> args;
8795         expr const &fn = get_app_args(val, args);
8796         lean_assert(args.size() == 2);
8797         unsigned n;
8798         lean_verify(is_llnf_uset(fn, n));
8799         return ir::mk_uset(to_var_id(args[0]), n, to_var_id(args[1]), b);
8800     }
8801
8802     ir::fn_body visit_proj(local_decl const &decl, ir::fn_body const &b) {
8803         expr val = *decl.get_value();
8804         unsigned i;
8805         lean_verify(is_llnf_proj(get_app_fn(val), i));
8806         ir::expr v = ir::mk_proj(i, to_var_id(app_arg(val)));
8807         return mk_vdecl(decl, v, b);
8808     }
8809
8810     ir::fn_body visit_sproj(local_decl const &decl, ir::fn_body const &b) {
8811         expr val = *decl.get_value();
8812         unsigned sz, n, offset;
8813         lean_verify(is_llnf_sproj(get_app_fn(val), sz, n, offset));
8814         ir::expr v = ir::mk_sproj(n, offset, to_var_id(app_arg(val)));
8815         return mk_vdecl(decl, v, b);
8816     }
8817
8818     ir::fn_body visit_fproj(local_decl const &decl, ir::fn_body const &b) {
8819         expr val = *decl.get_value();
```

```
8820         unsigned n, offset;
8821         lean_verify(is_llnf_fproj(get_app_fn(val), n, offset));
8822         ir::expr v = ir::mk_sproj(n, offset, to_var_id(app_arg(val)));
8823         return mk_vdecl(decl, v, b);
8824     }
8825
8826     ir::fn_body visit_uproj(local_decl const &decl, ir::fn_body const &b) {
8827         expr val = *decl.get_value();
8828         unsigned n;
8829         lean_verify(is_llnf_uproj(get_app_fn(val), n));
8830         ir::expr v = ir::mk_uproj(n, to_var_id(app_arg(val)));
8831         return mk_vdecl(decl, v, b);
8832     }
8833
8834     ir::fn_body visit_decl(local_decl const &decl, ir::fn_body const &b) {
8835         expr val = *decl.get_value();
8836         lean_assert(!is_fvar(val));
8837         if (is_lit(val)) {
8838             return visit_lit(decl, b);
8839         } else if (optional<nat> const &n = get_num_lit_ext(val)) {
8840             ir::type type = to_ir_type(decl.get_type());
8841             ir::expr val = ir::mk_num_lit(*n);
8842             return ir::mk_vdecl(to_var_id(decl), type, val, b);
8843         } else if (is_lambda(val)) {
8844             return visit_jp(decl, b);
8845         } else {
8846             expr const &fn = get_app_fn(val);
8847             if (is_llnf_cnstr(fn))
8848                 return visit_ctor(decl, b);
8849             else if (is_enf_unreachable(fn))
8850                 return ir::mk_unreachable();
8851             else if (is_llnf_apply(fn))
8852                 return visit_app(decl, b);
8853             else if (is_llnf_closure(fn))
8854                 return visit_papp(decl, b);
8855             else if (is_llnf_sset(fn))
8856                 return visit_sset(decl, b);
8857             else if (is_llnf_fset(fn))
8858                 return visit_fset(decl, b);
8859             else if (is_llnf_uset(fn))
8860                 return visit_uset(decl, b);
8861             else if (is_llnf_proj(fn))
8862                 return visit_proj(decl, b);
8863             else if (is_llnf_sproj(fn))
8864                 return visit_sproj(decl, b);
8865             else if (is_llnf_fproj(fn))
8866                 return visit_fproj(decl, b);
8867             else if (is_llnf_uproj(fn))
8868                 return visit_uproj(decl, b);
8869             else if (is_constant(fn))
8870                 return visit_fapp(decl, b);
8871             else
8872                 lean_unreachable();
8873         }
8874     }
8875
8876     ir::fn_body to_ir_fn_body(expr e) {
8877         buffer<expr> fvars;
8878         buffer<expr> subst;
8879         while (is_let(e)) {
8880             expr type = let_type(e);
8881             lean_assert(!has_loose_bvars(type));
8882             expr val =
8883                 instantiate_rev(let_value(e), subst.size(), subst.data());
8884             if (is_fvar(val) || is_enf_neutral(val)) {
8885                 /* Eliminate `x := y` and `x := _neutral` declarations */
8886                 subst.push_back(val);
8887             } else {
8888                 name n = next_name();
8889                 expr new_fvar = m_lctx.mk_local_decl(ngen(), n, type, val);
```

```
                fvars.push_back(new_fvar);
                expr const &op = get_app_fn(val);
                if (is_llnf_sset(op) || is_llnf_fset(op) || is_llnf_uset(op)) {
                    /* In the Lean IR, sset and uset are instructions that
                     * perform destructive updates. */
                    subst.push_back(app_arg(app_fn(val)));
                } else {
                    subst.push_back(new_fvar);
                }
            }
            e = let_body(e);
        }
        e = instantiate_rev(e, subst.size(), subst.data());
        ir::fn_body r = visit_terminal(e);
        unsigned i = fvars.size();
        while (i > 0) {
            --i;
            expr const &fvar = fvars[i];
            local_decl decl = m_lctx.get_local_decl(fvar);
            r = visit_decl(decl, r);
        }
        return r;
    }

    ir::decl to_ir_decl(comp_decl const &d) {
        name const &fn = d.fst();
        expr e = d.snd();
        buffer<ir::param> xs;
        ir::fn_body b = visit_lambda(e, xs);
        ir::type type =
            to_ir_result_type(get_constant_ll_type(env(), fn), xs.size());
        return ir::mk_decl(fn, xs, type, b);
    }

   public:
    to_ir_fn(environment const &env) : m_st(env) {}

    ir::decl operator()(comp_decl const &d) { return to_ir_decl(d); }

    /* Convert extern constant into a IR.Decl */
    ir::decl operator()(name const &fn) {
        buffer<bool> borrow;
        bool dummy;
        get_extern_borrowed_info(env(), fn, borrow, dummy);
        buffer<ir::param> xs;
        unsigned arity = *get_extern_constant_arity(env(), fn);
        expr type = get_constant_ll_type(env(), fn);
        for (unsigned i = 0; i < arity; i++) {
            lean_assert(is_pi(type));
            xs.push_back(ir::mk_param(
                ir::var_id(i), to_ir_type(binding_domain(type)), borrow[i]));
            type = binding_body(type);
        }
        ir::type result_type = to_ir_type(type);
        extern_attr_data_value attr = *get_extern_attr_data(env(), fn);
        return ir::mk_extern_decl(fn, xs, result_type, attr);
    }
};

namespace ir {
decl to_ir_decl(environment const &env, comp_decl const &d) {
    return to_ir_fn(env)(d);
}

/*
@[export lean.ir.compile_core]
def compile (env : Environment) (opts : Options) (decls : Array Decl) : Log ×
(Except String Environment) :=
*/
environment compile(environment const &env, options const &opts,
```

```
8960                     comp_decls const &decls) {
8961     buffer<decl> ir_decls;
8962     for (comp_decl const &decl : decls) {
8963         lean_trace(name({"compiler", "lambda_pure"}),
8964                    tout() << ">> " << decl.fst() << "\n"
8965                           << decl.snd() << "\n";);
8966         ir_decls.push_back(to_ir_decl(env, decl));
8967     }
8968     object *r = lean_ir_compile(env.to_obj_arg(), opts.to_obj_arg(),
8969                                 to_array(ir_decls));
8970     object *log = cnstr_get(r, 0);
8971     if (array_size(log) > 0) {
8972         inc(log);
8973         object *str = lean_ir_log_to_string(log);
8974         tout() << string_cstr(str);
8975         dec_ref(str);
8976     }
8977     object *v = cnstr_get(r, 1);
8978     if (cnstr_tag(v) == 0) {
8979         string_ref error(cnstr_get(v, 0), true);
8980         dec_ref(r);
8981         throw exception(error.data());
8982     } else {
8983         environment new_env(cnstr_get(v, 0), true);
8984         dec_ref(r);
8985         return new_env;
8986     }
8987 }
8988
8989 /*
8990 @[export lean_ir_add_boxed_version]
8991 def addBoxedVersion (env : Environment) (decl : Decl) : Except String
8992 Environment :=
8993 */
8994 extern "C" object *lean_ir_add_boxed_version(object *env, object *decl);
8995 environment add_boxed_version(environment const &env, decl const &d) {
8996     object *v = lean_ir_add_boxed_version(env.to_obj_arg(), d.to_obj_arg());
8997     if (cnstr_tag(v) == 0) {
8998         string_ref error(cnstr_get(v, 0), true);
8999         dec_ref(v);
9000         throw exception(error.data());
9001     } else {
9002         environment new_env(cnstr_get(v, 0), true);
9003         dec_ref(v);
9004         return new_env;
9005     }
9006 }
9007
9008 environment add_extern(environment const &env, name const &fn) {
9009     decl d = to_ir_fn(env)(fn);
9010     environment new_env = ir::add_decl(env, d);
9011     return add_boxed_version(new_env, d);
9012 }
9013
9014 extern "C" object *lean_add_extern(object *env, object *fn) {
9015     try {
9016         environment new_env = add_extern(environment(env), name(fn));
9017         return mk_except_ok(new_env);
9018     } catch (exception &ex) {
9019         // throw; // We use to uncomment this line when debugging weird bugs in
9020         // the Lean/C++ interface.
9021         return mk_except_error_string(ex.what());
9022     }
9023 }
9024
9025 extern "C" object *lean_ir_emit_c(object *env, object *mod_name);
9026
9027 string_ref emit_c(environment const &env, name const &mod_name) {
9028     object *r = lean_ir_emit_c(env.to_obj_arg(), mod_name.to_obj_arg());
9029     string_ref s(cnstr_get(r, 0), true);
```

```
    if (cnstr_tag(r) == 0) {
        dec_ref(r);
        throw exception(s.to_std_string());
    } else {
        dec_ref(r);
        return s;
    }
}

/*
inductive CtorFieldInfo
| irrelevant
| object (i : Nat)
| usize  (i : Nat)
| scalar (sz : Nat) (offset : Nat) (type : IRType)

structure CtorLayout :=
(cidx       : Nat)
(fieldInfo  : List CtorFieldInfo)
(numObjs    : Nat)
(numUSize   : Nat)
(scalarSize : Nat)
*/
object_ref to_object_ref(cnstr_info const &info) {
    buffer<object_ref> fields;
    for (field_info const &finfo : info.m_field_info) {
        switch (finfo.m_kind) {
            case field_info::Irrelevant:
                fields.push_back(object_ref(box(0)));
                break;
            case field_info::Object:
                fields.push_back(mk_cnstr(1, nat(finfo.m_idx)));
                break;
            case field_info::USize:
                fields.push_back(mk_cnstr(2, nat(finfo.m_idx)));
                break;
            case field_info::Scalar:
                fields.push_back(mk_cnstr(
                    3, nat(finfo.m_size), nat(finfo.m_offset),
                    object_ref(ir::box_type(to_ir_type(finfo.m_type)))));
                break;
        }
    }
    return mk_cnstr(0, nat(info.m_cidx), list_ref<object_ref>(fields),
                    nat(info.m_num_objs), nat(info.m_num_usizes),
                    nat(info.m_scalar_sz));
}

extern "C" object *lean_ir_get_ctor_layout(object *env0, object *ctor_name0) {
    environment const &env = TO_REF(environment, env0);
    name const &ctor_name = TO_REF(name, ctor_name0);
    type_checker::state st(env);
    try {
        cnstr_info info = get_cnstr_info(st, ctor_name);
        return mk_except_ok(to_object_ref(info));
    } catch (exception &ex) {
        return mk_except_error_string(ex.what());
    }
}
}  // namespace ir

void initialize_ir() { ir::irrelevant_arg = box(1); }

void finalize_ir() {}
}  // namespace lean
// ::::::::::::::
// compiler/ir_interpreter.cpp
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
```

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

Author: Sebastian Ullrich

A simple interpreter for evaluating λRC IR code.

Motivation
==========

Even with a JIT compiler, we still have a need for a simpler interpreter on
platforms LLVM JIT does not support (i.e. WebAssembly). Because this is mostly
an edge case, we strive for simplicity instead of performance and thus reuse the
existing compiler IR instead of inventing something like a new bytecode format.

Implementation
==============

The interpreter mainly consists of a homogeneous stack of `value`s, which are
either unboxed values or pointers to boxed objects. The IR type system tells us
which union member is active at any time. IR variables are mapped to stack slots
by adding the current base pointer to the variable index. Further stacks are
used for storing join points and call stack metadata. The interpreted IR is
taken directly from the environment. Whenever possible, we try to switch to
native code by checking for the mangled symbol via dlsym/GetProcAddress, which
is also how we can call external functions (which only works if the file
declaring them has already been compiled). We always call the "boxed" versions
of native functions, which have a (relatively) homogeneous ABI that we can use
without runtime code generation; see also `call/lookup_symbol` below.

*/
#include <string>
#include <vector>
#ifdef LEAN_WINDOWS
#include <windows.h>
#undef ERROR  // thanks, wingdi.h
#else
#include <dlfcn.h>
#endif
#include <lean/apply.h>
#include <lean/flet.h>
#include <lean/interrupt.h>
#include <lean/io.h>

#include "library/compiler/init_attribute.h"
#include "library/compiler/ir.h"
#include "library/time_task.h"
#include "library/trace.h"
#include "util/array_ref.h"
#include "util/nat.h"
#include "util/option_declarations.h"
#include "util/option_ref.h"

#ifndef LEAN_DEFAULT_INTERPRETER_PREFER_NATIVE
#if LEAN_IS_STAGE0 == 1
// We already set `-Dinterpreter.prefer_native=false` in stdlib.make, but also
// set it here as a default when we use stage 0 in the editor
#define LEAN_DEFAULT_INTERPRETER_PREFER_NATIVE false
#else
#define LEAN_DEFAULT_INTERPRETER_PREFER_NATIVE true
#endif
#endif

namespace lean {
namespace ir {
// C++ wrappers of Lean data types

typedef object_ref lit_val;
typedef object_ref ctor_info;

type to_type(object *obj) {
```

```
    if (!is_scalar(obj))
        throw exception("unsupported IRType");
    else
        return static_cast<type>(unbox(obj));
}
type cnstr_get_type(object_ref const &o, unsigned i) {
    return to_type(cnstr_get(o.raw(), i));
}

bool arg_is_irrelevant(arg const &a) { return is_scalar(a.raw()); }
var_id const &arg_var_id(arg const &a) {
    lean_assert(!arg_is_irrelevant(a));
    return cnstr_get_ref_t<var_id>(a, 0);
}

enum class lit_val_kind { Num, Str };
lit_val_kind lit_val_tag(lit_val const &l) {
    return static_cast<lit_val_kind>(cnstr_tag(l.raw()));
}
nat const &lit_val_num(lit_val const &l) {
    lean_assert(lit_val_tag(l) == lit_val_kind::Num);
    return cnstr_get_ref_t<nat>(l, 0);
}
string_ref const &lit_val_str(lit_val const &l) {
    lean_assert(lit_val_tag(l) == lit_val_kind::Str);
    return cnstr_get_ref_t<string_ref>(l, 0);
}

name const &ctor_info_name(ctor_info const &c) {
    return cnstr_get_ref_t<name>(c, 0);
}
nat const &ctor_info_tag(ctor_info const &c) {
    return cnstr_get_ref_t<nat>(c, 1);
}
nat const &ctor_info_size(ctor_info const &c) {
    return cnstr_get_ref_t<nat>(c, 2);
}
nat const &ctor_info_usize(ctor_info const &c) {
    return cnstr_get_ref_t<nat>(c, 3);
}
nat const &ctor_info_ssize(ctor_info const &c) {
    return cnstr_get_ref_t<nat>(c, 4);
}

/* Return the only Bool scalar field in an object that has `num_obj_fields`
 * object/usize fields */
static inline bool get_bool_field(object *o, unsigned num_obj_fields) {
    return cnstr_get_uint8(o, sizeof(void *) * num_obj_fields);
}

enum class expr_kind {
    Ctor,
    Reset,
    Reuse,
    Proj,
    UProj,
    SProj,
    FAp,
    PAp,
    Ap,
    Box,
    Unbox,
    Lit,
    IsShared,
    IsTaggedPtr
};
expr_kind expr_tag(expr const &e) {
    return static_cast<expr_kind>(cnstr_tag(e.raw()));
}
ctor_info const &expr_ctor_info(expr const &e) {
```

```
9240     lean_assert(expr_tag(e) == expr_kind::Ctor);
9241     return cnstr_get_ref_t<ctor_info>(e, 0);
9242 }
9243 array_ref<arg> const &expr_ctor_args(expr const &e) {
9244     lean_assert(expr_tag(e) == expr_kind::Ctor);
9245     return cnstr_get_ref_t<array_ref<arg>>(e, 1);
9246 }
9247 nat const &expr_reset_num_objs(expr const &e) {
9248     lean_assert(expr_tag(e) == expr_kind::Reset);
9249     return cnstr_get_ref_t<nat>(e, 0);
9250 }
9251 var_id const &expr_reset_obj(expr const &e) {
9252     lean_assert(expr_tag(e) == expr_kind::Reset);
9253     return cnstr_get_ref_t<var_id>(e, 1);
9254 }
9255 var_id const &expr_reuse_obj(expr const &e) {
9256     lean_assert(expr_tag(e) == expr_kind::Reuse);
9257     return cnstr_get_ref_t<var_id>(e, 0);
9258 }
9259 ctor_info const &expr_reuse_ctor(expr const &e) {
9260     lean_assert(expr_tag(e) == expr_kind::Reuse);
9261     return cnstr_get_ref_t<ctor_info>(e, 1);
9262 }
9263 bool expr_reuse_update_header(expr const &e) {
9264     lean_assert(expr_tag(e) == expr_kind::Reuse);
9265     return get_bool_field(e.raw(), 3);
9266 }
9267 array_ref<arg> const &expr_reuse_args(expr const &e) {
9268     lean_assert(expr_tag(e) == expr_kind::Reuse);
9269     return cnstr_get_ref_t<array_ref<arg>>(e, 2);
9270 }
9271 nat const &expr_proj_idx(expr const &e) {
9272     lean_assert(expr_tag(e) == expr_kind::Proj);
9273     return cnstr_get_ref_t<nat>(e, 0);
9274 }
9275 var_id const &expr_proj_obj(expr const &e) {
9276     lean_assert(expr_tag(e) == expr_kind::Proj);
9277     return cnstr_get_ref_t<var_id>(e, 1);
9278 }
9279 nat const &expr_uproj_idx(expr const &e) {
9280     lean_assert(expr_tag(e) == expr_kind::UProj);
9281     return cnstr_get_ref_t<nat>(e, 0);
9282 }
9283 var_id const &expr_uproj_obj(expr const &e) {
9284     lean_assert(expr_tag(e) == expr_kind::UProj);
9285     return cnstr_get_ref_t<var_id>(e, 1);
9286 }
9287 nat const &expr_sproj_idx(expr const &e) {
9288     lean_assert(expr_tag(e) == expr_kind::SProj);
9289     return cnstr_get_ref_t<nat>(e, 0);
9290 }
9291 nat const &expr_sproj_offset(expr const &e) {
9292     lean_assert(expr_tag(e) == expr_kind::SProj);
9293     return cnstr_get_ref_t<nat>(e, 1);
9294 }
9295 var_id const &expr_sproj_obj(expr const &e) {
9296     lean_assert(expr_tag(e) == expr_kind::SProj);
9297     return cnstr_get_ref_t<var_id>(e, 2);
9298 }
9299 fun_id const &expr_fap_fun(expr const &e) {
9300     lean_assert(expr_tag(e) == expr_kind::FAp);
9301     return cnstr_get_ref_t<fun_id>(e, 0);
9302 }
9303 array_ref<arg> const &expr_fap_args(expr const &e) {
9304     lean_assert(expr_tag(e) == expr_kind::FAp);
9305     return cnstr_get_ref_t<array_ref<arg>>(e, 1);
9306 }
9307 fun_id const &expr_pap_fun(expr const &e) {
9308     lean_assert(expr_tag(e) == expr_kind::PAp);
9309     return cnstr_get_ref_t<name>(e, 0);
```

```
9310 }
9311 array_ref<arg> const &expr_pap_args(expr const &e) {
9312     lean_assert(expr_tag(e) == expr_kind::PAp);
9313     return cnstr_get_ref_t<array_ref<arg>>(e, 1);
9314 }
9315 var_id const &expr_ap_fun(expr const &e) {
9316     lean_assert(expr_tag(e) == expr_kind::Ap);
9317     return cnstr_get_ref_t<var_id>(e, 0);
9318 }
9319 array_ref<arg> const &expr_ap_args(expr const &e) {
9320     lean_assert(expr_tag(e) == expr_kind::Ap);
9321     return cnstr_get_ref_t<array_ref<arg>>(e, 1);
9322 }
9323 type expr_box_type(expr const &e) {
9324     lean_assert(expr_tag(e) == expr_kind::Box);
9325     return cnstr_get_type(e, 0);
9326 }
9327 var_id const &expr_box_obj(expr const &e) {
9328     lean_assert(expr_tag(e) == expr_kind::Box);
9329     return cnstr_get_ref_t<var_id>(e, 1);
9330 }
9331 var_id const &expr_unbox_obj(expr const &e) {
9332     lean_assert(expr_tag(e) == expr_kind::Unbox);
9333     return cnstr_get_ref_t<var_id>(e, 0);
9334 }
9335 lit_val const &expr_lit_val(expr const &e) {
9336     lean_assert(expr_tag(e) == expr_kind::Lit);
9337     return cnstr_get_ref_t<lit_val>(e, 0);
9338 }
9339 var_id const &expr_is_shared_obj(expr const &e) {
9340     lean_assert(expr_tag(e) == expr_kind::IsShared);
9341     return cnstr_get_ref_t<var_id>(e, 0);
9342 }
9343 var_id const &expr_is_tagged_ptr_obj(expr const &e) {
9344     lean_assert(expr_tag(e) == expr_kind::IsTaggedPtr);
9345     return cnstr_get_ref_t<var_id>(e, 0);
9346 }
9347
9348 typedef object_ref param;
9349 var_id const &param_var(param const &p) {
9350     return cnstr_get_ref_t<var_id>(p, 0);
9351 }
9352 bool param_borrow(param const &p) { return get_bool_field(p.raw(), 2); }
9353 type param_type(param const &p) { return cnstr_get_type(p, 1); }
9354
9355 typedef object_ref alt_core;
9356 enum class alt_core_kind { Ctor, Default };
9357 alt_core_kind alt_core_tag(alt_core const &a) {
9358     return static_cast<alt_core_kind>(cnstr_tag(a.raw()));
9359 }
9360 ctor_info const &alt_core_ctor_info(alt_core const &a) {
9361     lean_assert(alt_core_tag(a) == alt_core_kind::Ctor);
9362     return cnstr_get_ref_t<ctor_info>(a, 0);
9363 }
9364 fn_body const &alt_core_ctor_cont(alt_core const &a) {
9365     lean_assert(alt_core_tag(a) == alt_core_kind::Ctor);
9366     return cnstr_get_ref_t<fn_body>(a, 1);
9367 }
9368 fn_body const &alt_core_default_cont(alt_core const &a) {
9369     lean_assert(alt_core_tag(a) == alt_core_kind::Default);
9370     return cnstr_get_ref_t<fn_body>(a, 0);
9371 }
9372
9373 enum class fn_body_kind {
9374     VDecl,
9375     JDecl,
9376     Set,
9377     SetTag,
9378     USet,
9379     SSet,
```

```
9380     Inc,
9381     Dec,
9382     Del,
9383     MData,
9384     Case,
9385     Ret,
9386     Jmp,
9387     Unreachable
9388 };
9389 fn_body_kind fn_body_tag(fn_body const &a) {
9390     return is_scalar(a.raw()) ? static_cast<fn_body_kind>(unbox(a.raw()))
9391                               : static_cast<fn_body_kind>(cnstr_tag(a.raw()));
9392 }
9393 var_id const &fn_body_vdecl_var(fn_body const &b) {
9394     lean_assert(fn_body_tag(b) == fn_body_kind::VDecl);
9395     return cnstr_get_ref_t<var_id>(b, 0);
9396 }
9397 type fn_body_vdecl_type(fn_body const &b) {
9398     lean_assert(fn_body_tag(b) == fn_body_kind::VDecl);
9399     return cnstr_get_type(b, 1);
9400 }
9401 expr const &fn_body_vdecl_expr(fn_body const &b) {
9402     lean_assert(fn_body_tag(b) == fn_body_kind::VDecl);
9403     return cnstr_get_ref_t<expr>(b, 2);
9404 }
9405 fn_body const &fn_body_vdecl_cont(fn_body const &b) {
9406     lean_assert(fn_body_tag(b) == fn_body_kind::VDecl);
9407     return cnstr_get_ref_t<fn_body>(b, 3);
9408 }
9409 jp_id const &fn_body_jdecl_id(fn_body const &b) {
9410     lean_assert(fn_body_tag(b) == fn_body_kind::JDecl);
9411     return cnstr_get_ref_t<jp_id>(b, 0);
9412 }
9413 array_ref<param> const &fn_body_jdecl_params(fn_body const &b) {
9414     lean_assert(fn_body_tag(b) == fn_body_kind::JDecl);
9415     return cnstr_get_ref_t<array_ref<param>>(b, 1);
9416 }
9417 fn_body const &fn_body_jdecl_body(fn_body const &b) {
9418     lean_assert(fn_body_tag(b) == fn_body_kind::JDecl);
9419     return cnstr_get_ref_t<fn_body>(b, 2);
9420 }
9421 fn_body const &fn_body_jdecl_cont(fn_body const &b) {
9422     lean_assert(fn_body_tag(b) == fn_body_kind::JDecl);
9423     return cnstr_get_ref_t<fn_body>(b, 3);
9424 }
9425 var_id const &fn_body_set_var(fn_body const &b) {
9426     lean_assert(fn_body_tag(b) == fn_body_kind::Set);
9427     return cnstr_get_ref_t<var_id>(b, 0);
9428 }
9429 nat const &fn_body_set_idx(fn_body const &b) {
9430     lean_assert(fn_body_tag(b) == fn_body_kind::Set);
9431     return cnstr_get_ref_t<nat>(b, 1);
9432 }
9433 arg const &fn_body_set_arg(fn_body const &b) {
9434     lean_assert(fn_body_tag(b) == fn_body_kind::Set);
9435     return cnstr_get_ref_t<arg>(b, 2);
9436 }
9437 fn_body const &fn_body_set_cont(fn_body const &b) {
9438     lean_assert(fn_body_tag(b) == fn_body_kind::Set);
9439     return cnstr_get_ref_t<fn_body>(b, 3);
9440 }
9441 var_id const &fn_body_set_tag_var(fn_body const &b) {
9442     lean_assert(fn_body_tag(b) == fn_body_kind::SetTag);
9443     return cnstr_get_ref_t<var_id>(b, 0);
9444 }
9445 nat const &fn_body_set_tag_cidx(fn_body const &b) {
9446     lean_assert(fn_body_tag(b) == fn_body_kind::SetTag);
9447     return cnstr_get_ref_t<nat>(b, 1);
9448 }
9449 fn_body const &fn_body_set_tag_cont(fn_body const &b) {
```

```
9450     lean_assert(fn_body_tag(b) == fn_body_kind::SetTag);
9451     return cnstr_get_ref_t<fn_body>(b, 2);
9452 }
9453 var_id const &fn_body_uset_target(fn_body const &b) {
9454     lean_assert(fn_body_tag(b) == fn_body_kind::USet);
9455     return cnstr_get_ref_t<var_id>(b, 0);
9456 }
9457 nat const &fn_body_uset_idx(fn_body const &b) {
9458     lean_assert(fn_body_tag(b) == fn_body_kind::USet);
9459     return cnstr_get_ref_t<nat>(b, 1);
9460 }
9461 var_id const &fn_body_uset_source(fn_body const &b) {
9462     lean_assert(fn_body_tag(b) == fn_body_kind::USet);
9463     return cnstr_get_ref_t<var_id>(b, 2);
9464 }
9465 fn_body const &fn_body_uset_cont(fn_body const &b) {
9466     lean_assert(fn_body_tag(b) == fn_body_kind::USet);
9467     return cnstr_get_ref_t<fn_body>(b, 3);
9468 }
9469 var_id const &fn_body_sset_target(fn_body const &b) {
9470     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9471     return cnstr_get_ref_t<var_id>(b, 0);
9472 }
9473 nat const &fn_body_sset_idx(fn_body const &b) {
9474     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9475     return cnstr_get_ref_t<nat>(b, 1);
9476 }
9477 nat const &fn_body_sset_offset(fn_body const &b) {
9478     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9479     return cnstr_get_ref_t<nat>(b, 2);
9480 }
9481 var_id const &fn_body_sset_source(fn_body const &b) {
9482     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9483     return cnstr_get_ref_t<var_id>(b, 3);
9484 }
9485 type fn_body_sset_type(fn_body const &b) {
9486     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9487     return cnstr_get_type(b, 4);
9488 }
9489 fn_body const &fn_body_sset_cont(fn_body const &b) {
9490     lean_assert(fn_body_tag(b) == fn_body_kind::SSet);
9491     return cnstr_get_ref_t<fn_body>(b, 5);
9492 }
9493 var_id const &fn_body_inc_var(fn_body const &b) {
9494     lean_assert(fn_body_tag(b) == fn_body_kind::Inc);
9495     return cnstr_get_ref_t<var_id>(b, 0);
9496 }
9497 nat const &fn_body_inc_val(fn_body const &b) {
9498     lean_assert(fn_body_tag(b) == fn_body_kind::Inc);
9499     return cnstr_get_ref_t<nat>(b, 1);
9500 }
9501 bool fn_body_inc_maybe_scalar(fn_body const &b) {
9502     lean_assert(fn_body_tag(b) == fn_body_kind::Inc);
9503     return get_bool_field(b.raw(), 3);
9504 }
9505 fn_body const &fn_body_inc_cont(fn_body const &b) {
9506     lean_assert(fn_body_tag(b) == fn_body_kind::Inc);
9507     return cnstr_get_ref_t<fn_body>(b, 2);
9508 }
9509 var_id const &fn_body_dec_var(fn_body const &b) {
9510     lean_assert(fn_body_tag(b) == fn_body_kind::Dec);
9511     return cnstr_get_ref_t<var_id>(b, 0);
9512 }
9513 nat const &fn_body_dec_val(fn_body const &b) {
9514     lean_assert(fn_body_tag(b) == fn_body_kind::Dec);
9515     return cnstr_get_ref_t<nat>(b, 1);
9516 }
9517 bool fn_body_dec_maybe_scalar(fn_body const &b) {
9518     lean_assert(fn_body_tag(b) == fn_body_kind::Dec);
9519     return get_bool_field(b.raw(), 3);
```

```
9520 }
9521 fn_body const &fn_body_dec_cont(fn_body const &b) {
9522     lean_assert(fn_body_tag(b) == fn_body_kind::Dec);
9523     return cnstr_get_ref_t<fn_body>(b, 2);
9524 }
9525 var_id const &fn_body_del_var(fn_body const &b) {
9526     lean_assert(fn_body_tag(b) == fn_body_kind::Del);
9527     return cnstr_get_ref_t<var_id>(b, 0);
9528 }
9529 fn_body const &fn_body_del_cont(fn_body const &b) {
9530     lean_assert(fn_body_tag(b) == fn_body_kind::Del);
9531     return cnstr_get_ref_t<fn_body>(b, 1);
9532 }
9533 object_ref const &fn_body_mdata_data(fn_body const &b) {
9534     lean_assert(fn_body_tag(b) == fn_body_kind::MData);
9535     return cnstr_get_ref_t<object_ref>(b, 0);
9536 }
9537 fn_body const &fn_body_mdata_cont(fn_body const &b) {
9538     lean_assert(fn_body_tag(b) == fn_body_kind::MData);
9539     return cnstr_get_ref_t<fn_body>(b, 1);
9540 }
9541 name const &fn_body_case_tid(fn_body const &b) {
9542     lean_assert(fn_body_tag(b) == fn_body_kind::Case);
9543     return cnstr_get_ref_t<name>(b, 0);
9544 }
9545 var_id const &fn_body_case_var(fn_body const &b) {
9546     lean_assert(fn_body_tag(b) == fn_body_kind::Case);
9547     return cnstr_get_ref_t<var_id>(b, 1);
9548 }
9549 type fn_body_case_var_type(fn_body const &b) {
9550     lean_assert(fn_body_tag(b) == fn_body_kind::Case);
9551     return cnstr_get_type(b, 2);
9552 }
9553 array_ref<alt_core> const &fn_body_case_alts(fn_body const &b) {
9554     lean_assert(fn_body_tag(b) == fn_body_kind::Case);
9555     return cnstr_get_ref_t<array_ref<alt_core>>(b, 3);
9556 }
9557 arg const &fn_body_ret_arg(fn_body const &b) {
9558     lean_assert(fn_body_tag(b) == fn_body_kind::Ret);
9559     return cnstr_get_ref_t<arg>(b, 0);
9560 }
9561 jp_id const &fn_body_jmp_jp(fn_body const &b) {
9562     lean_assert(fn_body_tag(b) == fn_body_kind::Jmp);
9563     return cnstr_get_ref_t<jp_id>(b, 0);
9564 }
9565 array_ref<arg> const &fn_body_jmp_args(fn_body const &b) {
9566     lean_assert(fn_body_tag(b) == fn_body_kind::Jmp);
9567     return cnstr_get_ref_t<array_ref<arg>>(b, 1);
9568 }
9569
9570 typedef object_ref decl;
9571 enum class decl_kind { Fun, Extern };
9572 decl_kind decl_tag(decl const &a) {
9573     return is_scalar(a.raw()) ? static_cast<decl_kind>(unbox(a.raw()))
9574                               : static_cast<decl_kind>(cnstr_tag(a.raw()));
9575 }
9576 fun_id const &decl_fun_id(decl const &b) {
9577     return cnstr_get_ref_t<fun_id>(b, 0);
9578 }
9579 array_ref<param> const &decl_params(decl const &b) {
9580     return cnstr_get_ref_t<array_ref<param>>(b, 1);
9581 }
9582 type decl_type(decl const &b) { return cnstr_get_type(b, 2); }
9583 fn_body const &decl_fun_body(decl const &b) {
9584     lean_assert(decl_tag(b) == decl_kind::Fun);
9585     return cnstr_get_ref_t<fn_body>(b, 3);
9586 }
9587
9588 extern "C" object *lean_ir_find_env_decl(object *env, object *n);
9589 option_ref<decl> find_ir_decl(environment const &env, name const &n) {
```

```
9590     return option_ref<decl>(
9591         lean_ir_find_env_decl(env.to_obj_arg(), n.to_obj_arg()));
9592 }
9593
9594 static string_ref *g_mangle_prefix = nullptr;
9595 static string_ref *g_boxed_suffix = nullptr;
9596 static string_ref *g_boxed_mangled_suffix = nullptr;
9597 static name *g_interpreter_prefer_native = nullptr;
9598
9599 // constants (lacking native declarations) initialized by `lean_run_init`
9600 static name_map<object *> *g_init_globals;
9601
9602 // reuse the compiler's name mangling to compute native symbol names
9603 extern "C" object *lean_name_mangle(object *n, object *pre);
9604 string_ref name_mangle(name const &n, string_ref const &pre) {
9605     return string_ref(lean_name_mangle(n.to_obj_arg(), pre.to_obj_arg()));
9606 }
9607
9608 extern "C" object *lean_ir_format_fn_body_head(object *b);
9609 format format_fn_body_head(fn_body const &b) {
9610     return format(lean_ir_format_fn_body_head(b.to_obj_arg()));
9611 }
9612
9613 static bool type_is_scalar(type t) {
9614     return t != type::Object && t != type::TObject && t != type::Irrelevant;
9615 }
9616
9617 extern "C" object *lean_get_regular_init_fn_name_for(object *env, object *fn);
9618 optional<name> get_regular_init_fn_name_for(environment const &env,
9619                                             name const &n) {
9620     return to_optional<name>(
9621         lean_get_regular_init_fn_name_for(env.to_obj_arg(), n.to_obj_arg()));
9622 }
9623
9624 extern "C" object *lean_get_builtin_init_fn_name_for(object *env, object *fn);
9625 optional<name> get_builtin_init_fn_name_for(environment const &env,
9626                                             name const &n) {
9627     return to_optional<name>(
9628         lean_get_builtin_init_fn_name_for(env.to_obj_arg(), n.to_obj_arg()));
9629 }
9630
9631 /** \brief Value stored in an interpreter variable slot */
9632 union value {
9633     // NOTE: the IR type system guarantees that we always access the active
9634     // union member
9635     uint64 m_num;  // big enough for any unboxed integral type
9636     static_assert(sizeof(size_t) <= sizeof(uint64),
9637                   "uint64 should be the largest unboxed type");  // NOLINT
9638     double m_float;
9639     object *m_obj;
9640
9641     value() {}
9642     // too convenient to make explicit
9643     value(uint64 num) : m_num(num) {}
9644     value(object *o) : m_obj(o) {}
9645
9646     // would overlap with `value(uint64)` as a constructor
9647     static value from_float(double f) {
9648         value v;
9649         v.m_float = f;
9650         return v;
9651     }
9652 };
9653
9654 object *box_t(value v, type t) {
9655     switch (t) {
9656         case type::Float:
9657             return box_float(v.m_float);
9658         case type::UInt8:
9659             return box(v.m_num);
```

```
9660         case type::UInt16:
9661             return box(v.m_num);
9662         case type::UInt32:
9663             return box_uint32(v.m_num);
9664         case type::UInt64:
9665             return box_uint64(v.m_num);
9666         case type::USize:
9667             return box_size_t(v.m_num);
9668         case type::Object:
9669         case type::TObject:
9670         case type::Irrelevant:
9671             return v.m_obj;
9672     }
9673 }
9674
9675 value unbox_t(object *o, type t) {
9676     switch (t) {
9677         case type::Float:
9678             return value::from_float(unbox_float(o));
9679         case type::UInt8:
9680             return unbox(o);
9681         case type::UInt16:
9682             return unbox(o);
9683         case type::UInt32:
9684             return unbox_uint32(o);
9685         case type::UInt64:
9686             return unbox_uint64(o);
9687         case type::USize:
9688             return unbox_size_t(o);
9689         default:
9690             lean_unreachable();
9691     }
9692 }
9693
9694 /** \pre Very simple debug output of arbitrary values, should be extended. */
9695 void print_value(std::ostream &ios, value const &v, type t) {
9696     if (t == type::Float) {
9697         ios << v.m_float;
9698     } else if (type_is_scalar(t)) {
9699         ios << v.m_num;
9700     } else {
9701         if (is_scalar(v.m_obj)) {
9702             ios << unbox(v.m_obj);
9703         } else if (v.m_obj == nullptr) {
9704             ios << "0x0";  // confusingly printed as "0" by the default
9705                            // operator<<
9706         } else {
9707             // merely following the trace of object addresses is surprisingly
9708             // helpful for debugging
9709             ios << v.m_obj;
9710         }
9711     }
9712 }
9713
9714 void *lookup_symbol_in_cur_exe(char const *sym) {
9715 #ifdef LEAN_WINDOWS
9716     return reinterpret_cast<void *>(
9717         GetProcAddress(GetModuleHandle(nullptr), sym));
9718 #else
9719     return dlsym(RTLD_DEFAULT, sym);
9720 #endif
9721 }
9722
9723 class interpreter;
9724 LEAN_THREAD_PTR(interpreter, g_interpreter);
9725
9726 class interpreter {
9727     // stack of IR variable slots
9728     std::vector<value> m_arg_stack;
9729     // stack of join points
```

```
9730     std::vector<fn_body const *> m_jp_stack;
9731     struct frame {
9732         name m_fn;
9733         // base pointers into the stack above
9734         size_t m_arg_bp;
9735         size_t m_jp_bp;
9736 
9737         frame(name const &mFn, size_t mArgBp, size_t mJpBp)
9738             : m_fn(mFn), m_arg_bp(mArgBp), m_jp_bp(mJpBp) {}
9739     };
9740     std::vector<frame> m_call_stack;
9741     environment const &m_env;
9742     options const &m_opts;
9743     // if `false`, use IR code where possible
9744     bool m_prefer_native;
9745     struct constant_cache_entry {
9746         bool m_is_scalar;
9747         value m_val;
9748     };
9749     // caches values of nullary functions ("constants")
9750     name_map<constant_cache_entry> m_constant_cache;
9751     struct symbol_cache_entry {
9752         decl m_decl;
9753         // symbol address; `nullptr` if function does not have native code
9754         void *m_addr;
9755         // true iff we chose the boxed version of a function where the IR uses
9756         // the unboxed version
9757         bool m_boxed;
9758     };
9759     // caches symbol lookup successes _and_ failures
9760     name_map<symbol_cache_entry> m_symbol_cache;
9761 
9762     /** \brief Get current stack frame */
9763     inline frame &get_frame() { return m_call_stack.back(); }
9764 
9765     /** \brief Get reference to stack slot of IR variable */
9766     inline value &var(var_id const &v) {
9767         // variables are 1-indexed
9768         size_t i = get_frame().m_arg_bp + v.get_small_value() - 1;
9769         // we don't know the frame size (unless we do an additional IR pass), so
9770         // we extend it dynamically
9771         if (i >= m_arg_stack.size()) {
9772             m_arg_stack.resize(i + 1);
9773         }
9774         return m_arg_stack[i];
9775     }
9776 
9777    public:
9778     template <class T>
9779     static inline T with_interpreter(environment const &env,
9780                                      options const &opts,
9781                                      std::function<T(interpreter &)> const &f) {
9782         if (g_interpreter && is_eqp(g_interpreter->m_env, env) &&
9783             is_eqp(g_interpreter->m_opts, opts)) {
9784             return f(*g_interpreter);
9785         } else {
9786             // We changed threads or the closure was stored and called in a
9787             // different context.
9788             time_task t("interpretation", opts);
9789             scope_trace_env scope_trace(env, opts);
9790             // the caches contain data from the Environment, so we cannot reuse
9791             // them when changing it
9792             interpreter interp(env, opts);
9793             flet<interpreter *> fl(g_interpreter, &interp);
9794             return f(interp);
9795         }
9796     }
9797 
9798    private:
9799     value eval_arg(arg const &a) {
```

```
9800         // an "irrelevant" argument is type- or proof-erased; we can use an
9801         // arbitrary value for it
9802         return arg_is_irrelevant(a) ? box(0) : var(arg_var_id(a));
9803     }
9804
9805     /** \brief Allocate constructor object with given tag and arguments */
9806     object *alloc_ctor(ctor_info const &i, array_ref<arg> const &args) {
9807         size_t tag = ctor_info_tag(i).get_small_value();
9808         // number of boxed object fields
9809         size_t size = ctor_info_size(i).get_small_value();
9810         // number of unboxed USize fields (whose byte size the IR is ignorant
9811         // of)
9812         size_t usize = ctor_info_usize(i).get_small_value();
9813         // byte size of all other unboxed fields
9814         size_t ssize = ctor_info_ssize(i).get_small_value();
9815         if (size == 0 && usize == 0 && ssize == 0) {
9816             // a constructor without data is optimized to a tagged pointer
9817             return box(tag);
9818         } else {
9819             object *o = alloc_cnstr(tag, size, usize * sizeof(void *) + ssize);
9820             for (size_t i = 0; i < args.size(); i++) {
9821                 cnstr_set(o, i, eval_arg(args[i]).m_obj);
9822             }
9823             return o;
9824         }
9825     }
9826
9827     /** \brief Return closure pointing to interpreter stub taking interpreter
9828        data, declaration to be called, and partially applied arguments. */
9829     object *mk_stub_closure(decl const &d, unsigned n, object **args) {
9830         unsigned cls_size = 3 + decl_params(d).size();
9831         object *cls = alloc_closure(get_stub(cls_size), cls_size, 3 + n);
9832         closure_set(cls, 0, m_env.to_obj_arg());
9833         closure_set(cls, 1, m_opts.to_obj_arg());
9834         closure_set(cls, 2, d.to_obj_arg());
9835         for (unsigned i = 0; i < n; i++) closure_set(cls, 3 + i, args[i]);
9836         return cls;
9837     }
9838
9839     value eval_expr(expr const &e, type t) {
9840         switch (expr_tag(e)) {
9841             case expr_kind::Ctor:
9842                 return value{alloc_ctor(expr_ctor_info(e), expr_ctor_args(e))};
9843             case expr_kind::Reset: {  // release fields if unique reference in
9844                                      // preparation for `Reuse` below
9845                 object *o = var(expr_reset_obj(e)).m_obj;
9846                 if (is_exclusive(o)) {
9847                     for (size_t i = 0;
9848                          i < expr_reset_num_objs(e).get_small_value(); i++) {
9849                         cnstr_release(o, i);
9850                     }
9851                     return o;
9852                 } else {
9853                     dec_ref(o);
9854                     return box(0);
9855                 }
9856             }
9857             case expr_kind::Reuse: {  // reuse dead allocation if possible
9858                 object *o = var(expr_reuse_obj(e)).m_obj;
9859                 // check if `Reset` above had a unique reference it consumed
9860                 if (is_scalar(o)) {
9861                     // fall back to regular allocation
9862                     return alloc_ctor(expr_reuse_ctor(e), expr_reuse_args(e));
9863                 } else {
9864                     // create new constructor object in-place
9865                     if (expr_reuse_update_header(e)) {
9866                         cnstr_set_tag(o, ctor_info_tag(expr_reuse_ctor(e))
9867                                                 .get_small_value());
9868                     }
9869                     for (size_t i = 0; i < expr_reuse_args(e).size(); i++) {
```

```
9870                         cnstr_set(o, i, eval_arg(expr_reuse_args(e)[i]).m_obj);
9871                     }
9872                     return o;
9873                 }
9874             }
9875             case expr_kind::Proj:  // object field access
9876                 return cnstr_get(var(expr_proj_obj(e)).m_obj,
9877                                  expr_proj_idx(e).get_small_value());
9878             case expr_kind::UProj:  // USize field access
9879                 return cnstr_get_usize(var(expr_uproj_obj(e)).m_obj,
9880                                        expr_uproj_idx(e).get_small_value());
9881             case expr_kind::SProj: {  // other unboxed field access
9882                 size_t offset =
9883                     expr_sproj_idx(e).get_small_value() * sizeof(void *) +
9884                     expr_sproj_offset(e).get_small_value();
9885                 object *o = var(expr_sproj_obj(e)).m_obj;
9886                 switch (t) {
9887                     case type::Float:
9888                         return value::from_float(cnstr_get_float(o, offset));
9889                     case type::UInt8:
9890                         return cnstr_get_uint8(o, offset);
9891                     case type::UInt16:
9892                         return cnstr_get_uint16(o, offset);
9893                     case type::UInt32:
9894                         return cnstr_get_uint32(o, offset);
9895                     case type::UInt64:
9896                         return cnstr_get_uint64(o, offset);
9897                     default:
9898                         throw exception("invalid instruction");
9899                 }
9900             }
9901             case expr_kind::FAp: {  // satured ("full") application of top-level
9902                                     // function
9903                 if (expr_fap_args(e).size()) {
9904                     return call(expr_fap_fun(e), expr_fap_args(e));
9905                 } else {
9906                     // nullary function ("constant")
9907                     return load(expr_fap_fun(e), t);
9908                 }
9909             }
9910             case expr_kind::PAp: {  // unsatured (partial) application of
9911                                     // top-level function
9912                 symbol_cache_entry sym = lookup_symbol(expr_pap_fun(e));
9913                 if (sym.m_addr) {
9914                     // point closure directly at native symbol
9915                     object *cls = alloc_closure(sym.m_addr,
9916                                                 decl_params(sym.m_decl).size(),
9917                                                 expr_pap_args(e).size());
9918                     for (unsigned i = 0; i < expr_pap_args(e).size(); i++) {
9919                         closure_set(cls, i,
9920                                     eval_arg(expr_pap_args(e)[i]).m_obj);
9921                     }
9922                     return cls;
9923                 } else {
9924                     // point closure at interpreter stub
9925                     object **args = static_cast<object **>(LEAN_ALLOCA(
9926                         expr_pap_args(e).size() * sizeof(object *)));  // NOLINT
9927                     for (size_t i = 0; i < expr_pap_args(e).size(); i++) {
9928                         args[i] = eval_arg(expr_pap_args(e)[i]).m_obj;
9929                     }
9930                     return mk_stub_closure(sym.m_decl, expr_pap_args(e).size(),
9931                                            args);
9932                 }
9933             }
9934             case expr_kind::Ap: {  // (saturated or unsatured) application of
9935                                    // closure; mostly handled by runtime
9936                 object **args = static_cast<object **>(LEAN_ALLOCA(
9937                     expr_ap_args(e).size() * sizeof(object *)));  // NOLINT
9938                 for (size_t i = 0; i < expr_ap_args(e).size(); i++) {
9939                     args[i] = eval_arg(expr_ap_args(e)[i]).m_obj;
```

```
9940                }
9941                object *r = apply_n(var(expr_ap_fun(e)).m_obj,
9942                                    expr_ap_args(e).size(), args);
9943                return r;
9944            }
9945            case expr_kind::Box:  // box unboxed value
9946                return box_t(var(expr_box_obj(e)).m_num, expr_box_type(e));
9947            case expr_kind::Unbox:  // unbox boxed value
9948                return unbox_t(var(expr_unbox_obj(e)).m_obj, t);
9949            case expr_kind::Lit:  // load numeric or string literal
9950                switch (lit_val_tag(expr_lit_val(e))) {
9951                    case lit_val_kind::Num: {
9952                        nat const &n = lit_val_num(expr_lit_val(e));
9953                        switch (t) {
9954                            case type::Float:
9955                                return value::from_float(
9956                                    lean_float_of_nat(n.raw()));
9957                            case type::UInt8:
9958                            case type::UInt16:
9959                            case type::UInt32:
9960                            case type::USize:
9961                                return lean_usize_of_nat(n.raw());
9962                            case type::UInt64:
9963                                return lean_uint64_of_nat(n.raw());
9964                            // `nat` literal
9965                            case type::Object:
9966                            case type::TObject:
9967                                return n.to_obj_arg();
9968                            default:
9969                                throw exception("invalid instruction");
9970                        }
9971                    }
9972                    case lit_val_kind::Str:
9973                        return lit_val_str(expr_lit_val(e)).to_obj_arg();
9974                }
9975            case expr_kind::IsShared:
9976                return !is_exclusive(var(expr_is_shared_obj(e)).m_obj);
9977            case expr_kind::IsTaggedPtr:
9978                return !is_scalar(var(expr_is_tagged_ptr_obj(e)).m_obj);
9979            default:
9980                throw exception(sstream()
9981                                << "unexpected instruction kind "
9982                                << static_cast<unsigned>(expr_tag(e)));
9983        }
9984    }
9985
9986    void check_system() {
9987        try {
9988            lean::check_system("interpreter");
9989        } catch (stack_space_exception &ex) {
9990            sstream ss;
9991            ss << ex.what() << "\n";
9992            ss << "interpreter stacktrace:\n";
9993            for (unsigned i = 0; i < m_call_stack.size(); i++) {
9994                ss << "#" << (i + 1) << " "
9995                   << m_call_stack[m_call_stack.size() - i - 1].m_fn << "\n";
9996            }
9997            throw throwable(ss);
9998        }
9999    }
10000
10001    value eval_body(fn_body const &b0) {
10002        check_system();
10003
10004        // make reference reassignable...
10005        std::reference_wrapper<fn_body const> b(b0);
10006        while (true) {
10007            DEBUG_CODE(lean_trace(name({"interpreter", "step"}),
10008                                  tout()
10009                                      << std::string(m_call_stack.size(), ' ')
```

```
10010                                   << format_fn_body_head(b) << "\n";);)
10011             switch (fn_body_tag(b)) {
10012                 case fn_body_kind::VDecl: {  // variable declaration
10013                     expr const &e = fn_body_vdecl_expr(b);
10014                     fn_body const &cont = fn_body_vdecl_cont(b);
10015                     // tail recursion?
10016                     if (expr_tag(e) == expr_kind::FAp &&
10017                         expr_fap_fun(e) == get_frame().m_fn &&
10018                         fn_body_tag(cont) == fn_body_kind::Ret &&
10019                         !arg_is_irrelevant(fn_body_ret_arg(cont)) &&
10020                         arg_var_id(fn_body_ret_arg(cont)) ==
10021                             fn_body_vdecl_var(b)) {
10022                         // tail recursion! copy argument values to parameter
10023                         // slots and reset `b`
10024                         array_ref<arg> const &args = expr_fap_args(e);
10025                         // argument and parameter slots may overlap, so first
10026                         // copy arguments to end of stack
10027                         size_t old_size = m_arg_stack.size();
10028                         for (const auto &arg : args) {
10029                             m_arg_stack.push_back(eval_arg(arg));
10030                         }
10031                         // now copy to parameter slots
10032                         for (size_t i = 0; i < args.size(); i++) {
10033                             m_arg_stack[get_frame().m_arg_bp + i] =
10034                                 m_arg_stack[old_size + i];
10035                         }
10036                         m_arg_stack.resize(get_frame().m_arg_bp + args.size());
10037                         b = b0;
10038                         check_system();
10039                         break;
10040                     }
10041                     value v =
10042                         eval_expr(fn_body_vdecl_expr(b), fn_body_vdecl_type(b));
10043                     // NOTE: `var` must be called *after* `eval_expr` because
10044                     // the stack may get resized and invalidate the pointer
10045                     var(fn_body_vdecl_var(b)) = v;
10046                     DEBUG_CODE(
10047                         lean_trace(
10048                             name({"interpreter", "step"}),
10049                             tout() << std::string(m_call_stack.size(), ' ')
10050                                    << "=> x_";
10051                             tout()
10052                             << fn_body_vdecl_var(b).get_small_value() << " = ";
10053                             print_value(tout(), var(fn_body_vdecl_var(b)),
10054                                         fn_body_vdecl_type(b));
10055                             tout() << "\n";);)
10056                     b = fn_body_vdecl_cont(b);
10057                     break;
10058                 }
10059                 case fn_body_kind::JDecl: {  // join-point declaration; store in
10060                                              // stack slot just like variables
10061                     size_t i = get_frame().m_jp_bp +
10062                                fn_body_jdecl_id(b).get_small_value();
10063                     if (i >= m_jp_stack.size()) {
10064                         m_jp_stack.resize(i + 1);
10065                     }
10066                     m_jp_stack[i] = &b.get();
10067                     b = fn_body_jdecl_cont(b);
10068                     break;
10069                 }
10070                 case fn_body_kind::Set: {  // set boxed field of unique
10071                                            // reference
10072                     object *o = var(fn_body_set_var(b)).m_obj;
10073                     lean_assert(is_exclusive(o));
10074                     cnstr_set(o, fn_body_set_idx(b).get_small_value(),
10075                               eval_arg(fn_body_set_arg(b)).m_obj);
10076                     b = fn_body_set_cont(b);
10077                     break;
10078                 }
10079                 case fn_body_kind::SetTag: {  // set constructor tag of unique
```

```
10080                                       // reference
10081                 object *o = var(fn_body_set_tag_var(b)).m_obj;
10082                 lean_assert(is_exclusive(o));
10083                 cnstr_set_tag(o, fn_body_set_tag_cidx(b).get_small_value());
10084                 b = fn_body_set_tag_cont(b);
10085                 break;
10086             }
10087             case fn_body_kind::USet: {  // set USize field of unique
10088                                         // reference
10089                 object *o = var(fn_body_uset_target(b)).m_obj;
10090                 lean_assert(is_exclusive(o));
10091                 cnstr_set_usize(o, fn_body_uset_idx(b).get_small_value(),
10092                                 var(fn_body_uset_source(b)).m_num);
10093                 b = fn_body_uset_cont(b);
10094                 break;
10095             }
10096             case fn_body_kind::SSet: {  // set other unboxed field of unique
10097                                         // reference
10098                 object *o = var(fn_body_sset_target(b)).m_obj;
10099                 size_t offset =
10100                     fn_body_sset_idx(b).get_small_value() * sizeof(void *) +
10101                     fn_body_sset_offset(b).get_small_value();
10102                 value v = var(fn_body_sset_source(b));
10103                 lean_assert(is_exclusive(o));
10104                 switch (fn_body_sset_type(b)) {
10105                     case type::Float:
10106                         cnstr_set_float(o, offset, v.m_float);
10107                         break;
10108                     case type::UInt8:
10109                         cnstr_set_uint8(o, offset, v.m_num);
10110                         break;
10111                     case type::UInt16:
10112                         cnstr_set_uint16(o, offset, v.m_num);
10113                         break;
10114                     case type::UInt32:
10115                         cnstr_set_uint32(o, offset, v.m_num);
10116                         break;
10117                     case type::UInt64:
10118                         cnstr_set_uint64(o, offset, v.m_num);
10119                         break;
10120                     default:
10121                         throw exception(sstream() << "invalid instruction");
10122                 }
10123                 b = fn_body_sset_cont(b);
10124                 break;
10125             }
10126             case fn_body_kind::Inc:  // increment reference counter
10127                 inc(var(fn_body_inc_var(b)).m_obj,
10128                     fn_body_inc_val(b).get_small_value());
10129                 b = fn_body_inc_cont(b);
10130                 break;
10131             case fn_body_kind::Dec: {  // decrement reference counter
10132                 size_t n = fn_body_dec_val(b).get_small_value();
10133                 for (size_t i = 0; i < n; i++) {
10134                     dec(var(fn_body_dec_var(b)).m_obj);
10135                 }
10136                 b = fn_body_dec_cont(b);
10137                 break;
10138             }
10139             case fn_body_kind::Del:  // delete object of unique reference
10140                 lean_free_object(var(fn_body_del_var(b)).m_obj);
10141                 b = fn_body_del_cont(b);
10142                 break;
10143             case fn_body_kind::MData:  // metadata; no-op
10144                 b = fn_body_mdata_cont(b);
10145                 break;
10146             case fn_body_kind::Case: {  // branch according to constructor
10147                                         // tag
10148                 array_ref<alt_core> const &alts = fn_body_case_alts(b);
10149                 unsigned tag;
```

```
10150                   value v = var(fn_body_case_var(b));
10151                   if (type_is_scalar(fn_body_case_var_type(b))) {
10152                       tag = v.m_num;
10153                   } else {
10154                       tag = lean_obj_tag(v.m_obj);
10155                   }
10156                   for (alt_core const &a : alts) {
10157                       switch (alt_core_tag(a)) {
10158                           case alt_core_kind::Ctor:
10159                               if (tag == ctor_info_tag(alt_core_ctor_info(a))
10160                                               .get_small_value()) {
10161                                   b = alt_core_ctor_cont(a);
10162                                   goto done;
10163                               }
10164                               break;
10165                           case alt_core_kind::Default:
10166                               b = alt_core_default_cont(a);
10167                               goto done;
10168                       }
10169                   }
10170                   throw exception("incomplete case");
10171               done:
10172                   break;
10173               }
10174               case fn_body_kind::Ret:
10175                   return eval_arg(fn_body_ret_arg(b));
10176               case fn_body_kind::Jmp: {  // jump to join-point
10177                   fn_body const &jp =
10178                       *m_jp_stack[get_frame().m_jp_bp +
10179                                   fn_body_jmp_jp(b).get_small_value()];
10180                   lean_assert(fn_body_jdecl_params(jp).size() ==
10181                               fn_body_jmp_args(b).size());
10182                   for (size_t i = 0; i < fn_body_jdecl_params(jp).size();
10183                        i++) {
10184                       var(param_var(fn_body_jdecl_params(jp)[i])) =
10185                           eval_arg(fn_body_jmp_args(b)[i]);
10186                   }
10187                   b = fn_body_jdecl_body(jp);
10188                   break;
10189               }
10190               case fn_body_kind::Unreachable:
10191                   throw exception("unreachable code");
10192           }
10193       }
10194   }
10195
10196   // specify argument base pointer explicitly because we've usually already
10197   // pushed some function arguments
10198   void push_frame(decl const &d, size_t arg_bp) {
10199       DEBUG_CODE({
10200           lean_trace(
10201               name({"interpreter", "call"}),
10202               tout() << std::string(m_call_stack.size(), ' ')
10203                      << decl_fun_id(d);
10204               for (size_t i = arg_bp; i < m_arg_stack.size(); i++) {
10205                   tout() << " ";
10206                   print_value(tout(), m_arg_stack[i],
10207                               param_type(decl_params(d)[i - arg_bp]));
10208               } tout()
10209               << "\n";);
10210       });
10211       m_call_stack.emplace_back(decl_fun_id(d), arg_bp, m_jp_stack.size());
10212   }
10213
10214   void pop_frame(value DEBUG_CODE(r), type DEBUG_CODE(t)) {
10215       m_arg_stack.resize(get_frame().m_arg_bp);
10216       m_jp_stack.resize(get_frame().m_jp_bp);
10217       m_call_stack.pop_back();
10218       DEBUG_CODE({
10219           lean_trace(name({"interpreter", "call"}),
```

```
10220                       tout() << std::string(m_call_stack.size(), ' ') << "=> ";
10221                       print_value(tout(), r, t); tout() << "\n";);
10222         });
10223     }
10224
10225     /** \brief Return cached lookup result for given unmangled function name in
10226      * the current binary. */
10227     symbol_cache_entry lookup_symbol(name const &fn) {
10228         if (symbol_cache_entry const *e = m_symbol_cache.find(fn)) {
10229             return *e;
10230         } else {
10231             symbol_cache_entry e_new{get_decl(fn), nullptr, false};
10232             if (m_prefer_native ||
10233                 decl_tag(e_new.m_decl) == decl_kind::Extern ||
10234                 has_init_attribute(m_env, fn)) {
10235                 string_ref mangled = name_mangle(fn, *g_mangle_prefix);
10236                 string_ref boxed_mangled(string_append(
10237                     mangled.to_obj_arg(), g_boxed_mangled_suffix->raw()));
10238                 // check for boxed version first
10239                 if (void *p_boxed =
10240                         lookup_symbol_in_cur_exe(boxed_mangled.data())) {
10241                     e_new.m_addr = p_boxed;
10242                     e_new.m_boxed = true;
10243                 } else if (void *p = lookup_symbol_in_cur_exe(mangled.data())) {
10244                     // if there is no boxed version, there are no unboxed
10245                     // parameters, so use default version
10246                     e_new.m_addr = p;
10247                 }
10248             }
10249             m_symbol_cache.insert(fn, e_new);
10250             return e_new;
10251         }
10252     }
10253
10254     /** \brief Retrieve Lean declaration from environment. */
10255     decl get_decl(name const &fn) {
10256         option_ref<decl> d = find_ir_decl(m_env, fn);
10257         if (!d) {
10258             throw exception(sstream() << "unknown declaration '" << fn << "'");
10259         }
10260         return d.get().value();
10261     }
10262
10263     /** \brief Evaluate nullary function ("constant"). */
10264     value load(name const &fn, type t) {
10265         if (constant_cache_entry const *cached = m_constant_cache.find(fn)) {
10266             if (!cached->m_is_scalar) {
10267                 inc(cached->m_val.m_obj);
10268             }
10269             return cached->m_val;
10270         }
10271         if (object *const *o = g_init_globals->find(fn)) {
10272             // persistent, so no `inc` needed
10273             return *o;
10274         }
10275
10276         if (get_regular_init_fn_name_for(m_env, fn)) {
10277             // We don't know whether `[init]` decls can be re-executed, so let's
10278             // not.
10279             throw exception(sstream()
10280                             << "cannot evaluate `[init]` declaration '" << fn
10281                             << "' in the same module");
10282         }
10283         symbol_cache_entry e = lookup_symbol(fn);
10284         if (e.m_addr) {
10285             // constants do not have boxed wrappers, but we'll survive
10286             switch (t) {
10287                 case type::Float:
10288                     return value::from_float(*static_cast<double *>(e.m_addr));
10289                 case type::UInt8:
```

```
10290                     return *static_cast<uint8 *>(e.m_addr);
10291                 case type::UInt16:
10292                     return *static_cast<uint16 *>(e.m_addr);
10293                 case type::UInt32:
10294                     return *static_cast<uint32 *>(e.m_addr);
10295                 case type::UInt64:
10296                     return *static_cast<uint64 *>(e.m_addr);
10297                 case type::USize:
10298                     return *static_cast<size_t *>(e.m_addr);
10299                 case type::Object:
10300                 case type::TObject:
10301                 case type::Irrelevant:
10302                     return *static_cast<object **>(e.m_addr);
10303             }
10304         } else {
10305             push_frame(e.m_decl, m_arg_stack.size());
10306             value r = eval_body(decl_fun_body(e.m_decl));
10307             pop_frame(r, decl_type(e.m_decl));
10308             if (!type_is_scalar(t)) {
10309                 inc(r.m_obj);
10310             }
10311             m_constant_cache.insert(fn,
10312                                     constant_cache_entry{type_is_scalar(t), r});
10313             return r;
10314         }
10315     }
10316
10317     value call(name const &fn, array_ref<arg> const &args) {
10318         size_t old_size = m_arg_stack.size();
10319         value r;
10320         symbol_cache_entry e = lookup_symbol(fn);
10321         if (e.m_addr) {
10322             object **args2 = static_cast<object **>(
10323                 LEAN_ALLOCA(args.size() * sizeof(object *)));  // NOLINT
10324             for (size_t i = 0; i < args.size(); i++) {
10325                 type t = param_type(decl_params(e.m_decl)[i]);
10326                 args2[i] = box_t(eval_arg(args[i]), t);
10327                 if (e.m_boxed && param_borrow(decl_params(e.m_decl)[i])) {
10328                     // NOTE: If we chose the boxed version where the IR chose
10329                     // the unboxed one, we need to manually increment originally
10330                     // borrowed parameters because the wrapper will decrement
10331                     // these after the call. Basically the wrapper is more
10332                     // homogeneous (removing both unboxed and borrowed
10333                     // parameters) than we would need in this instance.
10334                     inc(args2[i]);
10335                 }
10336             }
10337             push_frame(e.m_decl, old_size);
10338             object *o = curry(e.m_addr, args.size(), args2);
10339             type t = decl_type(e.m_decl);
10340             if (type_is_scalar(t)) {
10341                 lean_assert(e.m_boxed);
10342                 // NOTE: this unboxing does not exist in the IR, so we should
10343                 // manually consume `o`
10344                 r = unbox_t(o, t);
10345                 lean_dec(o);
10346             } else {
10347                 r = o;
10348             }
10349         } else {
10350             if (decl_tag(e.m_decl) == decl_kind::Extern) {
10351                 throw exception(sstream()
10352                                 << "could not find native implementation of "
10353                                    "external declaration '"
10354                                 << fn << "'");
10355             }
10356             // evaluate args in old stack frame
10357             for (const auto &arg : args) {
10358                 m_arg_stack.push_back(eval_arg(arg));
10359             }
```

```
10360             push_frame(e.m_decl, old_size);
10361             r = eval_body(decl_fun_body(e.m_decl));
10362         }
10363         pop_frame(r, decl_type(e.m_decl));
10364         return r;
10365     }
10366
10367     // closure stub
10368     object *stub_m(object **args) {
10369         decl d(args[2]);
10370         size_t old_size = m_arg_stack.size();
10371         for (size_t i = 0; i < decl_params(d).size(); i++) {
10372             m_arg_stack.push_back(args[3 + i]);
10373         }
10374         push_frame(d, old_size);
10375         object *r = eval_body(decl_fun_body(d)).m_obj;
10376         pop_frame(r, type::TObject);
10377         return r;
10378     }
10379
10380     // static closure stub
10381     static object *stub_m_aux(object **args) {
10382         environment env(args[0]);
10383         options opts(args[1]);
10384         return with_interpreter<object *>(env, opts, [&](interpreter &interp) {
10385             return interp.stub_m(args);
10386         });
10387     }
10388
10389     // python3 -c 'for i in range(1,17): print(f"    static object *
10390     // stub_{i}_aux(" + ", ".join([f"object * x_{j}" for j in range(1,i+1)]) +
10391     // ") { object * args[] = { " + ", ".join([f"x_{j}" for j in range(1,i+1)])
10392     // + " }; return interpreter::stub_m_aux(args); }")'
10393     static object *stub_1_aux(object *x_1) {
10394         object *args[] = {x_1};
10395         return interpreter::stub_m_aux(args);
10396     }
10397     static object *stub_2_aux(object *x_1, object *x_2) {
10398         object *args[] = {x_1, x_2};
10399         return interpreter::stub_m_aux(args);
10400     }
10401     static object *stub_3_aux(object *x_1, object *x_2, object *x_3) {
10402         object *args[] = {x_1, x_2, x_3};
10403         return interpreter::stub_m_aux(args);
10404     }
10405     static object *stub_4_aux(object *x_1, object *x_2, object *x_3,
10406                               object *x_4) {
10407         object *args[] = {x_1, x_2, x_3, x_4};
10408         return interpreter::stub_m_aux(args);
10409     }
10410     static object *stub_5_aux(object *x_1, object *x_2, object *x_3,
10411                               object *x_4, object *x_5) {
10412         object *args[] = {x_1, x_2, x_3, x_4, x_5};
10413         return interpreter::stub_m_aux(args);
10414     }
10415     static object *stub_6_aux(object *x_1, object *x_2, object *x_3,
10416                               object *x_4, object *x_5, object *x_6) {
10417         object *args[] = {x_1, x_2, x_3, x_4, x_5, x_6};
10418         return interpreter::stub_m_aux(args);
10419     }
10420     static object *stub_7_aux(object *x_1, object *x_2, object *x_3,
10421                               object *x_4, object *x_5, object *x_6,
10422                               object *x_7) {
10423         object *args[] = {x_1, x_2, x_3, x_4, x_5, x_6, x_7};
10424         return interpreter::stub_m_aux(args);
10425     }
10426     static object *stub_8_aux(object *x_1, object *x_2, object *x_3,
10427                               object *x_4, object *x_5, object *x_6,
10428                               object *x_7, object *x_8) {
10429         object *args[] = {x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8};
```

```
10430         return interpreter::stub_m_aux(args);
10431     }
10432     static object *stub_9_aux(object *x_1, object *x_2, object *x_3,
10433                               object *x_4, object *x_5, object *x_6,
10434                              object *x_7, object *x_8, object *x_9) {
10435         object *args[] = {x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9};
10436         return interpreter::stub_m_aux(args);
10437     }
10438     static object *stub_10_aux(object *x_1, object *x_2, object *x_3,
10439                                object *x_4, object *x_5, object *x_6,
10440                                object *x_7, object *x_8, object *x_9,
10441                                object *x_10) {
10442         object *args[] = {x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_10};
10443         return interpreter::stub_m_aux(args);
10444     }
10445     static object *stub_11_aux(object *x_1, object *x_2, object *x_3,
10446                                object *x_4, object *x_5, object *x_6,
10447                                object *x_7, object *x_8, object *x_9,
10448                                object *x_10, object *x_11) {
10449         object *args[] = {x_1, x_2, x_3, x_4,  x_5, x_6,
10450                           x_7, x_8, x_9, x_10, x_11};
10451         return interpreter::stub_m_aux(args);
10452     }
10453     static object *stub_12_aux(object *x_1, object *x_2, object *x_3,
10454                                object *x_4, object *x_5, object *x_6,
10455                                object *x_7, object *x_8, object *x_9,
10456                                object *x_10, object *x_11, object *x_12) {
10457         object *args[] = {x_1, x_2, x_3, x_4,  x_5,  x_6,
10458                           x_7, x_8, x_9, x_10, x_11, x_12};
10459         return interpreter::stub_m_aux(args);
10460     }
10461     static object *stub_13_aux(object *x_1, object *x_2, object *x_3,
10462                                object *x_4, object *x_5, object *x_6,
10463                                object *x_7, object *x_8, object *x_9,
10464                                object *x_10, object *x_11, object *x_12,
10465                                object *x_13) {
10466         object *args[] = {x_1, x_2, x_3,  x_4,  x_5,  x_6, x_7,
10467                           x_8, x_9, x_10, x_11, x_12, x_13};
10468         return interpreter::stub_m_aux(args);
10469     }
10470     static object *stub_14_aux(object *x_1, object *x_2, object *x_3,
10471                                object *x_4, object *x_5, object *x_6,
10472                                object *x_7, object *x_8, object *x_9,
10473                                object *x_10, object *x_11, object *x_12,
10474                                object *x_13, object *x_14) {
10475         object *args[] = {x_1, x_2, x_3,  x_4,  x_5,  x_6,  x_7,
10476                           x_8, x_9, x_10, x_11, x_12, x_13, x_14};
10477         return interpreter::stub_m_aux(args);
10478     }
10479     static object *stub_15_aux(object *x_1, object *x_2, object *x_3,
10480                                object *x_4, object *x_5, object *x_6,
10481                                object *x_7, object *x_8, object *x_9,
10482                                object *x_10, object *x_11, object *x_12,
10483                                object *x_13, object *x_14, object *x_15) {
10484         object *args[] = {x_1, x_2,  x_3,  x_4,  x_5,  x_6,  x_7, x_8,
10485                           x_9, x_10, x_11, x_12, x_13, x_14, x_15};
10486         return interpreter::stub_m_aux(args);
10487     }
10488     static object *stub_16_aux(object *x_1, object *x_2, object *x_3,
10489                                object *x_4, object *x_5, object *x_6,
10490                                object *x_7, object *x_8, object *x_9,
10491                                object *x_10, object *x_11, object *x_12,
10492                                object *x_13, object *x_14, object *x_15,
10493                                object *x_16) {
10494         object *args[] = {x_1, x_2,  x_3,  x_4,  x_5,  x_6,  x_7,  x_8,
10495                           x_9, x_10, x_11, x_12, x_13, x_14, x_15, x_16};
10496         return interpreter::stub_m_aux(args);
10497     }
10498
10499     void *get_stub(unsigned params) {
```

```
10500         switch (params) {
10501             case 0:
10502                 lean_unreachable();
10503             case 1:
10504                 return reinterpret_cast<void *>(stub_1_aux);
10505             case 2:
10506                 return reinterpret_cast<void *>(stub_2_aux);
10507             case 3:
10508                 return reinterpret_cast<void *>(stub_3_aux);
10509             case 4:
10510                 return reinterpret_cast<void *>(stub_4_aux);
10511             case 5:
10512                 return reinterpret_cast<void *>(stub_5_aux);
10513             case 6:
10514                 return reinterpret_cast<void *>(stub_6_aux);
10515             case 7:
10516                 return reinterpret_cast<void *>(stub_7_aux);
10517             case 8:
10518                 return reinterpret_cast<void *>(stub_8_aux);
10519             case 9:
10520                 return reinterpret_cast<void *>(stub_9_aux);
10521             case 10:
10522                 return reinterpret_cast<void *>(stub_10_aux);
10523             case 11:
10524                 return reinterpret_cast<void *>(stub_11_aux);
10525             case 12:
10526                 return reinterpret_cast<void *>(stub_12_aux);
10527             case 13:
10528                 return reinterpret_cast<void *>(stub_13_aux);
10529             case 14:
10530                 return reinterpret_cast<void *>(stub_14_aux);
10531             case 15:
10532                 return reinterpret_cast<void *>(stub_15_aux);
10533             case 16:
10534                 return reinterpret_cast<void *>(stub_16_aux);
10535             default:
10536                 return reinterpret_cast<void *>(stub_m_aux);
10537         }
10538     }
10539
10540     public:
10541     explicit interpreter(environment const &env, options const &opts)
10542         : m_env(env), m_opts(opts) {
10543         m_prefer_native = opts.get_bool(*g_interpreter_prefer_native,
10544                                         LEAN_DEFAULT_INTERPRETER_PREFER_NATIVE);
10545     }
10546
10547     ~interpreter() {
10548         for_each(m_constant_cache,
10549                  [](name const &, constant_cache_entry const &e) {
10550                      if (!e.m_is_scalar) {
10551                          dec(e.m_val.m_obj);
10552                      }
10553                  });
10554     }
10555
10556     /** A variant of `call` designed for external uses.
10557      *  * takes (owned) `object *`s instead of `arg`s.
10558      *  * supports under- and over-application.
10559      *  * supports "calling" (evaluating) nullary constants. */
10560     object *call_boxed(name const &fn, unsigned n, object **args) {
10561         symbol_cache_entry e = lookup_symbol(fn);
10562         unsigned arity = decl_params(e.m_decl).size();
10563         object *r;
10564         if (arity == 0) {
10565             r = box_t(load(fn, decl_type(e.m_decl)), decl_type(e.m_decl));
10566         } else {
10567             // First allocate a closure with zero fixed parameters. This is
10568             // slightly wasteful in the under-application case, but simpler to
10569             // handle.
```

```
10570             if (e.m_addr) {
10571                 // `lookup_symbol` always prefers the boxed version for compiled
10572                 // functions, so nothing to do here
10573                 r = alloc_closure(e.m_addr, arity, 0);
10574             } else {
10575                 // `lookup_symbol` does not prefer the boxed version for
10576                 // interpreted functions, so check manually.
10577                 decl d = e.m_decl;
10578                 if (option_ref<decl> d_boxed =
10579                         find_ir_decl(m_env, fn + *g_boxed_suffix)) {
10580                     d = *d_boxed.get();
10581                 }
10582                 r = mk_stub_closure(d, 0, nullptr);
10583             }
10584         }
10585         if (n > 0) {
10586             r = apply_n(r, n, args);
10587         }
10588         return r;
10589     }
10590
10591     uint32 run_main(int argc, char *argv[]) {
10592         decl d = get_decl("main");
10593         array_ref<param> const &params = decl_params(d);
10594         buffer<object *> args;
10595         if (params.size() == 2) {  // List String -> IO UInt32
10596             lean_object *in = lean_box(0);
10597             int i = argc;
10598             while (i > 0) {
10599                 i--;
10600                 lean_object *n = lean_alloc_ctor(1, 2, 0);
10601                 lean_ctor_set(n, 0, lean_mk_string(argv[i]));
10602                 lean_ctor_set(n, 1, in);
10603                 in = n;
10604             }
10605             args.push_back(in);
10606         } else {  // IO UInt32
10607             lean_assert(params.size() == 1);
10608         }
10609         object *w = io_mk_world();
10610         args.push_back(w);
10611         w = call_boxed("main", args.size(), &args[0]);
10612         if (io_result_is_ok(w)) {
10613             // NOTE: in an awesome hack, `IO Unit` works just as well because
10614             // `pure 0` and `pure ()` use the same representation
10615             int ret = unbox(io_result_get_value(w));
10616             dec_ref(w);
10617             return ret;
10618         } else {
10619             io_result_show_error(w);
10620             dec_ref(w);
10621             return 1;
10622         }
10623     }
10624
10625     object *run_init(name const &decl, name const &init_decl) {
10626         try {
10627             object *args[] = {io_mk_world()};
10628             object *r = call_boxed(init_decl, 1, args);
10629             if (io_result_is_ok(r)) {
10630                 object *o = io_result_get_value(r);
10631                 mark_persistent(o);
10632                 dec_ref(r);
10633                 symbol_cache_entry e = lookup_symbol(decl);
10634                 if (e.m_addr) {
10635                     *((object **)e.m_addr) = o;
10636                 } else {
10637                     g_init_globals->insert(decl, o);
10638                 }
10639                 return lean_io_result_mk_ok(box(0));
```

```
10640             } else {
10641                 return r;
10642             }
10643         } catch (exception &ex) {
10644             return io_result_mk_error(ex.what());
10645         }
10646     }
10647 };
10648
10649 extern "C" object *lean_decl_get_sorry_dep(object *env, object *n);
10650
10651 optional<name> get_sorry_dep(environment const &env, name const &n) {
10652     return option_ref<name>(
10653                lean_decl_get_sorry_dep(env.to_obj_arg(), n.to_obj_arg()))
10654         .get();
10655 }
10656
10657 object *run_boxed(environment const &env, options const &opts, name const &fn,
10658                   unsigned n, object **args) {
10659     if (get_sorry_dep(env, fn)) {
10660         throw exception(
10661             "cannot evaluate code because it uses 'sorry' and/or contains "
10662             "errors");
10663     }
10664     return interpreter::with_interpreter<object *>(
10665         env, opts,
10666         [&](interpreter &interp) { return interp.call_boxed(fn, n, args); });
10667 }
10668 uint32 run_main(environment const &env, options const &opts, int argv,
10669                 char *argc[]) {
10670     return interpreter::with_interpreter<uint32>(
10671         env, opts,
10672         [&](interpreter &interp) { return interp.run_main(argv, argc); });
10673 }
10674
10675 extern "C" object *lean_eval_const(object *env, object *opts, object *c) {
10676     try {
10677         return mk_cnstr(
10678                    1, run_boxed(TO_REF(environment, env), TO_REF(options, opts),
10679                                 TO_REF(name, c), 0, 0))
10680             .steal();
10681     } catch (exception &ex) {
10682         return mk_cnstr(0, string_ref(ex.what())).steal();
10683     }
10684 }
10685
10686 extern "C" object *lean_run_init(object *env, object *opts, object *decl,
10687                                  object *init_decl, object *) {
10688     return interpreter::with_interpreter<object *>(
10689         TO_REF(environment, env), TO_REF(options, opts),
10690         [&](interpreter &interp) {
10691             return interp.run_init(TO_REF(name, decl), TO_REF(name, init_decl));
10692         });
10693 }
10694 }  // namespace ir
10695
10696 void initialize_ir_interpreter() {
10697     ir::g_mangle_prefix = new string_ref("l_");
10698     mark_persistent(ir::g_mangle_prefix->raw());
10699     ir::g_boxed_suffix = new string_ref("_boxed");
10700     mark_persistent(ir::g_boxed_suffix->raw());
10701     ir::g_boxed_mangled_suffix = new string_ref("___boxed");
10702     mark_persistent(ir::g_boxed_mangled_suffix->raw());
10703     ir::g_interpreter_prefer_native =
10704         new name({"interpreter", "prefer_native"});
10705     ir::g_init_globals = new name_map<object *>();
10706     register_bool_option(
10707         *ir::g_interpreter_prefer_native,
10708         LEAN_DEFAULT_INTERPRETER_PREFER_NATIVE,
10709         "(interpreter) whether to use precompiled code where available");
```

```
10710     DEBUG_CODE({
10711         register_trace_class({"interpreter"});
10712         register_trace_class({"interpreter", "call"});
10713         register_trace_class({"interpreter", "step"});
10714     });
10715 }
10716
10717 void finalize_ir_interpreter() {
10718     delete ir::g_init_globals;
10719     delete ir::g_interpreter_prefer_native;
10720     delete ir::g_boxed_mangled_suffix;
10721     delete ir::g_boxed_suffix;
10722     delete ir::g_mangle_prefix;
10723 }
10724 }  // namespace lean
10725 // ::::::::::::::
10726 // compiler/lambda_lifting.cpp
10727 // ::::::::::::::
10728 /*
10729 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
10730 Released under Apache 2.0 license as described in the file LICENSE.
10731
10732 Author: Leonardo de Moura
10733 */
10734 #include <lean/flet.h>
10735
10736 #include <unordered_set>
10737
10738 #include "kernel/abstract.h"
10739 #include "kernel/for_each_fn.h"
10740 #include "kernel/instantiate.h"
10741 #include "library/compiler/closed_term_cache.h"
10742 #include "library/compiler/util.h"
10743 #include "library/trace.h"
10744
10745 namespace lean {
10746 name mk_lambda_lifting_name(name const &fn, unsigned idx) {
10747     name r(fn, "_lambda");
10748     return r.append_after(idx);
10749 }
10750
10751 bool is_lambda_lifting_name(name fn) {
10752     while (!fn.is_atomic()) {
10753         if (fn.is_string() &&
10754             strncmp(fn.get_string().data(), "_lambda", 7) == 0)
10755             return true;
10756         fn = fn.get_prefix();
10757     }
10758     return false;
10759 }
10760
10761 class lambda_lifting_fn {
10762     environment m_env;
10763     name_generator m_ngen;
10764     local_ctx m_lctx;
10765     buffer<comp_decl> m_new_decls;
10766     name m_base_name;
10767     unsigned m_next_idx{1};
10768
10769     typedef std::unordered_set<name, name_hash_fn> name_set;
10770
10771     environment const &env() { return m_env; }
10772
10773     name_generator &ngen() { return m_ngen; }
10774
10775     expr visit_lambda_core(expr e) {
10776         flet<local_ctx> save_lctx(m_lctx, m_lctx);
10777         buffer<expr> fvars;
10778         while (is_lambda(e)) {
10779             lean_assert(!has_loose_bvars(binding_domain(e)));
```

```
10780             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
10781                                                  binding_domain(e));
10782             fvars.push_back(new_fvar);
10783             e = binding_body(e);
10784         }
10785         expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
10786         return m_lctx.mk_lambda(fvars, r);
10787     }
10788
10789     expr visit_let(expr e) {
10790         flet<local_ctx> save_lctx(m_lctx, m_lctx);
10791         buffer<expr> fvars;
10792         while (is_let(e)) {
10793             lean_assert(!has_loose_bvars(let_type(e)));
10794             bool not_root = false;
10795             bool jp = is_join_point_name(let_name(e));
10796             expr new_val =
10797                 visit(instantiate_rev(let_value(e), fvars.size(), fvars.data()),
10798                       not_root, jp);
10799             expr new_fvar =
10800                 m_lctx.mk_local_decl(ngen(), let_name(e), let_type(e), new_val);
10801             fvars.push_back(new_fvar);
10802             e = let_body(e);
10803         }
10804         expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
10805         return m_lctx.mk_lambda(fvars, r);
10806     }
10807
10808     expr visit_cases_on(expr const &e) {
10809         lean_assert(is_cases_on_app(env(), e));
10810         buffer<expr> args;
10811         expr const &c = get_app_args(e, args);
10812         /* Remark: lambda lifting is applied after we have erased most type
10813            information,
10814            and `cases_on` applications have major premise and minor premises
10815            only. */
10816         for (unsigned i = 1; i < args.size(); i++) {
10817             args[i] = visit_lambda_core(args[i]);
10818         }
10819         return mk_app(c, args);
10820     }
10821
10822     expr visit_app(expr const &e) {
10823         if (is_cases_on_app(env(), e)) {
10824             return visit_cases_on(e);
10825         } else {
10826             return e;
10827         }
10828     }
10829
10830     void collect_fvars_core(expr const &e, name_set collected,
10831                             buffer<expr> &fvars, buffer<expr> &jps) {
10832         if (!has_fvar(e)) return;
10833         for_each(e, [&](expr const &x, unsigned) {
10834             if (!has_fvar(x)) return false;
10835             if (is_fvar(x)) {
10836                 if (collected.find(fvar_name(x)) == collected.end()) {
10837                     collected.insert(fvar_name(x));
10838                     local_decl d = m_lctx.get_local_decl(x);
10839                     /* We MUST copy any join point that lambda expression
10840                        depends on, and its dependencies. */
10841                     if (is_join_point_name(d.get_user_name())) {
10842                         collect_fvars_core(*d.get_value(), collected, fvars,
10843                                            jps);
10844                         jps.push_back(x);
10845                     } else {
10846                         fvars.push_back(x);
10847                     }
10848                 }
10849             }
```

```
10850             return true;
10851         });
10852     }
10853
10854     void collect_fvars(expr const &e, buffer<expr> &fvars, buffer<expr> &jps) {
10855         if (!has_fvar(e)) return;
10856         name_set collected;
10857         collect_fvars_core(e, collected, fvars, jps);
10858     }
10859
10860     /* Try to apply eta-reduction to reduce number of auxiliary declarations. */
10861     optional<expr> try_eta_reduction(expr const &e) {
10862         expr r = ::lean::try_eta(e);
10863         expr const &f = get_app_fn(r);
10864
10865         if (is_fvar(f)) return some_expr(r);
10866
10867         if (is_constant(f)) {
10868             name const &n = const_name(f);
10869             if (!is_constructor(env(), n) && !is_cases_on_recursor(env(), n))
10870                 return some_expr(r);
10871         }
10872         return none_expr();
10873     }
10874
10875     name next_name() {
10876         name r = mk_lambda_lifting_name(m_base_name, m_next_idx);
10877         m_next_idx++;
10878         return r;
10879     }
10880
10881     /* Given `e` of the form `fun xs, t`, create  `fun fvars xs, let jps in e`.
10882      */
10883     expr mk_lambda(buffer<expr> const &fvars, buffer<expr> const &jps, expr e) {
10884         flet<local_ctx> save_lctx(m_lctx, m_lctx);
10885         buffer<expr> xs;
10886         while (is_lambda(e)) {
10887             lean_assert(!has_loose_bvars(binding_domain(e)));
10888             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
10889                                                  binding_domain(e));
10890             xs.push_back(new_fvar);
10891             e = binding_body(e);
10892         }
10893         e = instantiate_rev(e, xs.size(), xs.data());
10894         e = abstract(e, jps.size(), jps.data());
10895         unsigned i = jps.size();
10896         while (i > 0) {
10897             --i;
10898             expr const &fvar = jps[i];
10899             local_decl decl = m_lctx.get_local_decl(fvar);
10900             lean_assert(is_join_point_name(decl.get_user_name()));
10901             lean_assert(!has_loose_bvars(decl.get_type()));
10902             expr val = abstract(*decl.get_value(), i, jps.data());
10903             e = ::lean::mk_let(decl.get_user_name(), decl.get_type(), val, e);
10904         }
10905         e = m_lctx.mk_lambda(xs, e);
10906         e = abstract(e, fvars.size(), fvars.data());
10907         i = fvars.size();
10908         while (i > 0) {
10909             --i;
10910             expr const &fvar = fvars[i];
10911             local_decl decl = m_lctx.get_local_decl(fvar);
10912             lean_assert(!has_loose_bvars(decl.get_type()));
10913             e = ::lean::mk_lambda(decl.get_user_name(), decl.get_type(), e);
10914         }
10915         return e;
10916     }
10917
10918     expr visit_lambda(expr e, bool root, bool join_point) {
10919         e = visit_lambda_core(e);
```

```
10920         if (root || join_point) return e;
10921         if (optional<expr> r = try_eta_reduction(e)) return *r;
10922         buffer<expr> fvars;
10923         buffer<expr> jps;
10924         collect_fvars(e, fvars, jps);
10925         e = mk_lambda(fvars, jps, e);
10926         name new_fn;
10927         if (optional<name> opt_new_fn = get_closed_term_name(m_env, e)) {
10928             new_fn = *opt_new_fn;
10929         } else {
10930             new_fn = next_name();
10931             m_new_decls.push_back(comp_decl(new_fn, e));
10932             m_env = cache_closed_term_name(m_env, e, new_fn);
10933         }
10934         return mk_app(mk_constant(new_fn), fvars);
10935     }
10936
10937     expr visit(expr const &e, bool root = false, bool join_point = false) {
10938         switch (e.kind()) {
10939             case expr_kind::App:
10940                 return visit_app(e);
10941             case expr_kind::Lambda:
10942                 return visit_lambda(e, root, join_point);
10943             case expr_kind::Let:
10944                 return visit_let(e);
10945             default:
10946                 return e;
10947         }
10948     }
10949
10950    public:
10951     lambda_lifting_fn(environment const &env) : m_env(env) {}
10952
10953     pair<environment, comp_decls> operator()(comp_decl const &cdecl) {
10954         m_base_name = cdecl.fst();
10955         expr r = visit(cdecl.snd(), true);
10956         comp_decl new_cdecl(cdecl.fst(), r);
10957         m_new_decls.push_back(new_cdecl);
10958         return mk_pair(m_env, comp_decls(m_new_decls));
10959     }
10960 };
10961
10962 pair<environment, comp_decls> lambda_lifting(environment const &env,
10963                                              comp_decl const &d) {
10964     return lambda_lifting_fn(env)(d);
10965 }
10966
10967 pair<environment, comp_decls> lambda_lifting(environment env,
10968                                              comp_decls const &ds) {
10969     comp_decls r;
10970     for (comp_decl const &d : ds) {
10971         comp_decls new_ds;
10972         std::tie(env, new_ds) = lambda_lifting(env, d);
10973         r = append(r, new_ds);
10974     }
10975     return mk_pair(env, r);
10976 }
10977 }  // namespace lean
10978 // ::::::::::::::
10979 // compiler/lcnf.cpp
10980 // ::::::::::::::
10981 /*
10982 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
10983 Released under Apache 2.0 license as described in the file LICENSE.
10984
10985 Author: Leonardo de Moura
10986 */
10987 #include <lean/flet.h>
10988 #include <lean/sstream.h>
10989
```

```
10990 #include <algorithm>
10991
10992 #include "kernel/inductive.h"
10993 #include "kernel/instantiate.h"
10994 #include "kernel/replace_fn.h"
10995 #include "kernel/type_checker.h"
10996 #include "library/aux_recursors.h"
10997 #include "library/compiler/implemented_by_attribute.h"
10998 #include "library/compiler/util.h"
10999 #include "library/constants.h"
11000 #include "library/expr_lt.h"
11001 #include "library/num.h"
11002 #include "library/projection.h"
11003 #include "library/util.h"
11004
11005 namespace lean {
11006 /*
11007 @[export lean_erase_macro_scopes]
11008 def Name.eraseMacroScopes (n : Name) : Name :=
11009 */
11010 extern "C" object *lean_erase_macro_scopes(object *n);
11011 name erase_macro_scopes(name const &n) {
11012     return name(lean_erase_macro_scopes(n.to_obj_arg()));
11013 }
11014 // This is a big HACK for detecting joinpoints created by the do notation
11015 bool is_do_notation_joinpoint(name const &n) {
11016     name n2 = erase_macro_scopes(n);
11017     return n2 != n && "do!jp";
11018 }
11019
11020 class to_lcnf_fn {
11021     typedef rb_expr_map<expr> cache;
11022     type_checker::state m_st;
11023     local_ctx m_lctx;
11024     cache m_cache;
11025     buffer<expr> m_fvars;
11026     name m_x;
11027     unsigned m_next_idx{1};
11028
11029    public:
11030     to_lcnf_fn(environment const &env, local_ctx const &lctx)
11031         : m_st(env), m_lctx(lctx), m_x("_x") {}
11032
11033     environment &env() { return m_st.env(); }
11034
11035     name_generator &ngen() { return m_st.ngen(); }
11036
11037     expr infer_type(expr const &e) {
11038         return type_checker(m_st, m_lctx).infer(e);
11039     }
11040
11041     expr whnf(expr const &e) { return type_checker(m_st, m_lctx).whnf(e); }
11042
11043     expr whnf_infer_type(expr const &e) {
11044         type_checker tc(m_st, m_lctx);
11045         return tc.whnf(tc.infer(e));
11046     }
11047
11048     static bool is_lc_proof(expr const &e) {
11049         return is_app_of(e, get_lc_proof_name());
11050     }
11051
11052     name next_name() {
11053         name r = m_x.append_after(m_next_idx);
11054         m_next_idx++;
11055         return r;
11056     }
11057
11058     expr mk_let_decl(expr const &e, bool root) {
11059         if (root) {
```

```
11060             return e;
11061         } else {
11062             expr type = cheap_beta_reduce(infer_type(e));
11063             /* Remark: we use `m_x.append_after(m_next_idx)` instead of
11064                `name(m_x, m_next_idx)` because the resulting name is confusing
11065                during debugging: it looks like a projection application. We
11066                should replace it with `name(m_x, m_next_idx)` when the compiler
11067                code gets more stable. */
11068             expr fvar = m_lctx.mk_local_decl(ngen(), next_name(), type, e);
11069             m_fvars.push_back(fvar);
11070             return fvar;
11071         }
11072     }
11073
11074     expr mk_let(unsigned old_fvars_size, expr const &body) {
11075         lean_assert(m_fvars.size() >= old_fvars_size);
11076         expr r = m_lctx.mk_lambda(m_fvars.size() - old_fvars_size,
11077                                   m_fvars.data() + old_fvars_size, body);
11078         m_fvars.shrink(old_fvars_size);
11079         return r;
11080     }
11081
11082     expr eta_expand(expr e, unsigned num_extra) {
11083         lean_assert(num_extra > 0);
11084         flet<local_ctx> save_lctx(m_lctx, m_lctx);
11085         buffer<expr> args;
11086         lean_assert(!is_lambda(e));
11087         expr e_type = whnf_infer_type(e);
11088         for (unsigned i = 0; i < num_extra; i++) {
11089             if (!is_pi(e_type)) {
11090                 throw exception(
11091                     "compiler error, unexpected type at LCNF conversion");
11092             }
11093             expr arg = m_lctx.mk_local_decl(ngen(), binding_name(e_type),
11094                                             binding_domain(e_type),
11095                                             binding_info(e_type));
11096             args.push_back(arg);
11097             e_type = whnf(instantiate(binding_body(e_type), arg));
11098         }
11099         return m_lctx.mk_lambda(args, mk_app(e, args));
11100     }
11101
11102     expr visit_projection(expr const &fn, projection_info const &pinfo,
11103                           buffer<expr> &args, bool root) {
11104         name const &k = pinfo.get_constructor();
11105         constructor_val k_val = env().get(k).to_constructor_val();
11106         name const &I_name = k_val.get_induct();
11107         if (is_runtime_builtin_type(I_name)) {
11108             /* We should not expand projections of runtime builtin types */
11109             return visit_app_default(fn, args, root);
11110         } else {
11111             constant_info info = env().get(const_name(fn));
11112             expr fn_val = instantiate_value_lparams(info, const_levels(fn));
11113             std::reverse(args.begin(), args.end());
11114             return visit(apply_beta(fn_val, args.size(), args.data()), root);
11115         }
11116     }
11117
11118     unsigned get_constructor_nfields(name const &n) {
11119         return env().get(n).to_constructor_val().get_nfields();
11120     }
11121
11122     /* Return true iff the motive is of the form `(fun is x, t)` where `t` does
11123        not depend on `is` or `x`, and `is x` has size `nindices + 1`. */
11124     bool is_nondep_elim(expr motive, unsigned nindices) {
11125         for (unsigned i = 0; i < nindices + 1; i++) {
11126             if (!is_lambda(motive)) return false;
11127             motive = binding_body(motive);
11128         }
11129         return !has_loose_bvars(motive);
```

```
11130     }
11131
11132     expr visit_cases_on(expr const &fn, buffer<expr> &args, bool root) {
11133         name const &rec_name = const_name(fn);
11134         levels const &rec_levels = const_levels(fn);
11135         name const &I_name = rec_name.get_prefix();
11136         lean_assert(is_inductive(env(), I_name));
11137         constant_info I_info = env().get(I_name);
11138         inductive_val I_val = I_info.to_inductive_val();
11139         unsigned nparams = I_val.get_nparams();
11140         names cnstrs = I_val.get_cnstrs();
11141         unsigned nminors = length(cnstrs);
11142         unsigned nindices = I_val.get_nindices();
11143         unsigned major_idx = nparams + 1 /* typeformer/motive */ + nindices;
11144         unsigned first_minor_idx = major_idx + 1;
11145         unsigned arity = first_minor_idx + nminors;
11146         if (args.size() < arity) {
11147             return visit(eta_expand(mk_app(fn, args), arity - args.size()),
11148                          root);
11149         } else if (args.size() > arity) {
11150             expr new_cases = visit(mk_app(fn, arity, args.data()), false);
11151             return visit(
11152                 mk_app(new_cases, args.size() - arity, args.data() + arity),
11153                 root);
11154         } else {
11155             for (unsigned i = 0; i < first_minor_idx; i++) {
11156                 args[i] = visit(args[i], false);
11157             }
11158             expr major = args[major_idx];
11159             lean_assert(first_minor_idx + nminors == arity);
11160             for (unsigned i = first_minor_idx; i < arity; i++) {
11161                 name cnstr_name = head(cnstrs);
11162                 cnstrs = tail(cnstrs);
11163                 expr minor = args[i];
11164                 unsigned num_fields = get_constructor_nfields(cnstr_name);
11165                 flet<local_ctx> save_lctx(m_lctx, m_lctx);
11166                 buffer<expr> minor_fvars;
11167                 unsigned j = 0;
11168                 while (is_lambda(minor) && j < num_fields) {
11169                     expr new_d =
11170                         instantiate_rev(binding_domain(minor),
11171                                         minor_fvars.size(), minor_fvars.data());
11172                     expr new_fvar =
11173                         m_lctx.mk_local_decl(ngen(), binding_name(minor), new_d,
11174                                              binding_info(minor));
11175                     minor_fvars.push_back(new_fvar);
11176                     minor = binding_body(minor);
11177                     j++;
11178                 }
11179                 minor = instantiate_rev(minor, minor_fvars.size(),
11180                                         minor_fvars.data());
11181                 if (j < num_fields) {
11182                     minor = eta_expand(minor, num_fields - j);
11183                     for (; j < num_fields; j++) {
11184                         expr new_fvar = m_lctx.mk_local_decl(
11185                             ngen(), binding_name(minor), binding_domain(minor),
11186                             binding_info(minor));
11187                         minor_fvars.push_back(new_fvar);
11188                         minor = instantiate(binding_body(minor), new_fvar);
11189                     }
11190                 }
11191                 flet<cache> save_cache(m_cache, m_cache);
11192                 unsigned old_fvars_size = m_fvars.size();
11193                 expr new_minor = visit(minor, true);
11194                 if (is_lambda(new_minor))
11195                     new_minor = mk_let_decl(new_minor, false);
11196                 new_minor = mk_let(old_fvars_size, new_minor);
11197                 /* Create a constructor application with the "fields" of the
11198                    minor premise. Then, replace `k` with major premise at
11199                    new_minor. This transformation is important for code like
```

```
                this:
                ```
                def foo : Expr -> Expr
                | (Expr.app f a) := f
                | e   := e
                ```
                The equation compiler will "complete" the wildcard case `e :=
                e` by expanding `e`.

                Remark: this transformation is only safe for non-dependent
                elimination. It may produce type incorrect terms otherwise.
                We ignore this issue in the compiler.

                Remark: we *must* redesign the equation compiler. This
                transformation may produce unexpected results. For example,
                we seldom want it for `Bool`. For example, we don't want `or`
                ```
                def or (x y : Bool) : Bool :=
                match x with
                | true := true
                | false := y
                ```
                to be transformed into
                ```
                def or (x y : Bool) : Bool :=
                match x with
                | true := x
                | false := y
                ```

                On the other hand, we want the transformation to be applied
                to:
                ```
                def flatten : Format → Format
                | nil   := nil
                | line   := text " "
                | f@(text _)   := f
                | (nest _ f)   := flatten f
                | (choice f _)   := flatten f
                | f@(compose true _ _)   := f -- If we don't apply the
                transformation, we will "re-create" `f` here | f@(compose
                false f₁ f₂) := compose true (flatten f₁) (flatten f₂)
                ```

                Summary: we need to make sure the equation compiler preserves
                the user intent, and then disable this transformation.

                For now, we don't apply this transformation for Bool when the
                minor premise is equal to the major. That is, we make sure we
                don't do it for `and`, `or`, etc.
            */
            expr k =
                mk_app(mk_app(mk_constant(cnstr_name, tail(rec_levels)),
                              nparams, args.data()),
                       minor_fvars);
            if (I_name != get_bool_name() || new_minor != k) {
                expr new_new_minor =
                    replace(new_minor, [&](expr const &e, unsigned) {
                        if (e == k)
                            return some_expr(major);
                        else
                            return none_expr();
                    });
                if (new_new_minor != new_minor)
                    new_minor = elim_trivial_let_decls(new_new_minor);
            }
            new_minor = m_lctx.mk_lambda(minor_fvars, new_minor);
            args[i] = new_minor;
        }
        return mk_let_decl(mk_app(fn, args), root);
```

```
11270         }
11271     }
11272
11273     expr lit_to_constructor(expr const &e) {
11274         if (is_nat_lit(e))
11275             return nat_lit_to_constructor(e);
11276         else if (is_string_lit(e))
11277             return string_lit_to_constructor(e);
11278         else
11279             return e;
11280     }
11281
11282     expr visit_no_confusion(expr const &fn, buffer<expr> &args, bool root) {
11283         name const &no_confusion_name = const_name(fn);
11284         name const &I_name = no_confusion_name.get_prefix();
11285         constant_info I_info = env().get(I_name);
11286         inductive_val I_val = I_info.to_inductive_val();
11287         unsigned nparams = I_val.get_nparams();
11288         unsigned nindices = I_val.get_nindices();
11289         unsigned basic_arity = nparams + nindices + 1 /* motive */ +
11290                                2 /* lhs/rhs */ + 1 /* equality */;
11291         if (args.size() < basic_arity) {
11292             return visit(
11293                 eta_expand(mk_app(fn, args), basic_arity - args.size()), root);
11294         }
11295         lean_assert(args.size() >= basic_arity);
11296         type_checker tc(m_st, m_lctx);
11297         expr lhs = tc.whnf(args[nparams + nindices + 1]);
11298         expr rhs = tc.whnf(args[nparams + nindices + 2]);
11299         lhs = lit_to_constructor(lhs);
11300         rhs = lit_to_constructor(rhs);
11301         optional<name> lhs_constructor = is_constructor_app(env(), lhs);
11302         optional<name> rhs_constructor = is_constructor_app(env(), rhs);
11303         if (!lhs_constructor || !rhs_constructor)
11304             throw exception(sstream()
11305                             << "compiler error, unsupported occurrence of '"
11306                             << no_confusion_name << "', constructors expected");
11307         if (lhs_constructor != rhs_constructor) {
11308             expr type = tc.whnf(tc.infer(mk_app(fn, args)));
11309             level lvl = sort_level(tc.ensure_type(type));
11310             return mk_let_decl(
11311                 mk_app(mk_constant(get_lc_unreachable_name(), {lvl}), type),
11312                 root);
11313         } else if (args.size() < basic_arity + 1 /* major */) {
11314             return visit(
11315                 eta_expand(mk_app(fn, args), basic_arity + 1 - args.size()),
11316                 root);
11317         } else {
11318             lean_assert(args.size() >= basic_arity + 1);
11319             unsigned major_idx = basic_arity;
11320             expr major = args[major_idx];
11321             unsigned nfields = get_constructor_nfields(*lhs_constructor);
11322             while (nfields > 0) {
11323                 if (!is_lambda(major)) major = eta_expand(major, nfields);
11324                 lean_assert(is_lambda(major));
11325                 expr type = binding_domain(major);
11326                 lean_assert(tc.is_prop(type));
11327                 expr proof = mk_app(mk_constant(get_lc_proof_name()), type);
11328                 major = instantiate(binding_body(major), proof);
11329                 nfields--;
11330             }
11331             expr new_e = mk_app(major, args.size() - major_idx - 1,
11332                                 args.data() + major_idx + 1);
11333             return visit(new_e, root);
11334         }
11335     }
11336
11337     expr visit_eq_rec(expr const &fn, buffer<expr> &args, bool root) {
11338         lean_assert(const_name(fn) == get_eq_rec_name() ||
11339                     const_name(fn) == get_eq_ndrec_name() ||
```

```
11340                   const_name(fn) == get_eq_cases_on_name() ||
11341                   const_name(fn) == get_eq_rec_on_name());
11342         if (args.size() < 6) {
11343             return visit(eta_expand(mk_app(fn, args), 6 - args.size()), root);
11344         } else {
11345             unsigned eq_rec_nargs = 6;
11346             unsigned minor_idx;
11347             if (const_name(fn) == get_eq_cases_on_name() ||
11348                 const_name(fn) == get_eq_rec_on_name())
11349                 minor_idx = 5;
11350             else
11351                 minor_idx = 3;
11352             type_checker tc(m_st, m_lctx);
11353             expr minor = args[minor_idx];
11354             /* Remark: this reduction may introduce a type incorrect term here
11355                since type of minor may not be definitionally equal to the type
11356                of `mk_app(fn, args)`. */
11357             expr new_e = minor;
11358             new_e = mk_app(new_e, args.size() - eq_rec_nargs,
11359                            args.data() + eq_rec_nargs);
11360             return visit(new_e, root);
11361         }
11362     }
11363
11364     expr visit_false_rec(expr const &fn, buffer<expr> &args, bool root) {
11365         if (args.size() < 2) {
11366             return visit(eta_expand(mk_app(fn, args), 2 - args.size()), root);
11367         } else {
11368             /* Remark: args.size() may be greater than 2, but
11369                (lc_unreachable a_1 ... a_n) is equivalent to (lc_unreachable) */
11370             expr type = infer_type(mk_app(fn, args));
11371             return mk_let_decl(mk_lc_unreachable(m_st, m_lctx, type), root);
11372         }
11373     }
11374
11375     expr visit_and_rec(expr const &fn, buffer<expr> &args, bool root) {
11376         lean_assert(const_name(fn) == get_and_rec_name() ||
11377                     const_name(fn) == get_and_cases_on_name());
11378         if (args.size() < 5) {
11379             return visit(eta_expand(mk_app(fn, args), 5 - args.size()), root);
11380         } else {
11381             expr a = args[0];
11382             expr b = args[1];
11383             expr pr_a = mk_app(mk_constant(get_lc_proof_name()), a);
11384             expr pr_b = mk_app(mk_constant(get_lc_proof_name()), b);
11385             expr minor;
11386             if (const_name(fn) == get_and_rec_name())
11387                 minor = args[3];
11388             else
11389                 minor = args[4];
11390             return visit(mk_app(minor, pr_a, pr_b), root);
11391         }
11392     }
11393
11394     expr visit_constructor(expr const &fn, buffer<expr> &args, bool root) {
11395         constructor_val cval = env().get(const_name(fn)).to_constructor_val();
11396         unsigned arity = cval.get_nparams() + cval.get_nfields();
11397         if (args.size() < arity) {
11398             return visit(eta_expand(mk_app(fn, args), arity - args.size()),
11399                          root);
11400         } else {
11401             return visit_app_default(fn, args, root);
11402         }
11403     }
11404
11405     bool should_create_let_decl(expr const &e, expr e_type) {
11406         switch (e.kind()) {
11407             case expr_kind::BVar:
11408             case expr_kind::MVar:
11409             case expr_kind::FVar:
```

```
11410             case expr_kind::Sort:
11411             case expr_kind::Const:
11412             case expr_kind::Lit:
11413             case expr_kind::Pi:
11414                 return false;
11415             default:
11416                 break;
11417         }
11418         if (is_lc_proof(e)) return false;
11419         if (is_irrelevant_type(m_st, m_lctx, e_type)) return false;
11420         return true;
11421     }
11422
11423     expr visit_app_default(expr const &fn, buffer<expr> &args, bool root) {
11424         if (args.empty()) {
11425             return fn;
11426         } else {
11427             for (expr &arg : args) {
11428                 arg = visit(arg, false);
11429             }
11430             return mk_let_decl(mk_app(fn, args), root);
11431         }
11432     }
11433
11434     expr visit_quot(expr const &fn, buffer<expr> &args, bool root) {
11435         constant_info info = env().get(const_name(fn));
11436         lean_assert(info.is_quot());
11437         unsigned arity = 0;
11438         switch (info.to_quot_val().get_quot_kind()) {
11439             case quot_kind::Type:
11440             case quot_kind::Ind:
11441                 return visit_app_default(fn, args, root);
11442             case quot_kind::Mk:
11443                 arity = 3;
11444                 break;
11445             case quot_kind::Lift:
11446                 arity = 6;
11447                 break;
11448         }
11449         if (args.size() < arity) {
11450             return visit(eta_expand(mk_app(fn, args), arity - args.size()),
11451                          root);
11452         } else {
11453             return visit_app_default(fn, args, root);
11454         }
11455     }
11456
11457     expr visit_constant_core(expr fn, buffer<expr> &args, bool root) {
11458         if (const_name(fn) == get_and_rec_name() ||
11459             const_name(fn) == get_and_cases_on_name()) {
11460             return visit_and_rec(fn, args, root);
11461         } else if (const_name(fn) == get_eq_rec_name() ||
11462                    const_name(fn) == get_eq_ndrec_name() ||
11463                    const_name(fn) == get_eq_cases_on_name() ||
11464                    const_name(fn) == get_eq_rec_on_name()) {
11465             return visit_eq_rec(fn, args, root);
11466         } else if (const_name(fn) == get_false_rec_name() ||
11467                    const_name(fn) == get_false_cases_on_name() ||
11468                    const_name(fn) == get_empty_rec_name() ||
11469                    const_name(fn) == get_empty_cases_on_name()) {
11470             return visit_false_rec(fn, args, root);
11471         } else if (is_cases_on_recursor(env(), const_name(fn))) {
11472             return visit_cases_on(fn, args, root);
11473         } else if (optional<projection_info> pinfo =
11474                        get_projection_info(env(), const_name(fn))) {
11475             return visit_projection(fn, *pinfo, args, root);
11476         } else if (is_no_confusion(env(), const_name(fn))) {
11477             return visit_no_confusion(fn, args, root);
11478         } else if (is_constructor(env(), const_name(fn))) {
11479             return visit_constructor(fn, args, root);
```

```
11480         } else if (optional<name> n = is_unsafe_rec_name(const_name(fn))) {
11481             fn = mk_constant(*n, const_levels(fn));
11482             return visit_app_default(fn, args, root);
11483         } else if (is_quot_primitive(env(), const_name(fn))) {
11484             return visit_quot(fn, args, root);
11485         } else if (optional<name> n =
11486                        get_implemented_by_attribute(env(), const_name(fn))) {
11487             return visit_app_default(mk_constant(*n, const_levels(fn)), args,
11488                                      root);
11489         } else {
11490             return visit_app_default(fn, args, root);
11491         }
11492     }
11493
11494     expr visit_constant(expr const &e, bool root) {
11495         if (const_name(e) == get_nat_zero_name()) {
11496             return mk_lit(literal(nat(0)));
11497         } else {
11498             buffer<expr> args;
11499             return visit_constant_core(e, args, root);
11500         }
11501     }
11502
11503     expr visit_app(expr const &e, bool root) {
11504         /* TODO(Leo): remove after we add support for literals in the front-end
11505          */
11506         if (optional<mpz> v = to_num(e)) {
11507             expr type = whnf_infer_type(e);
11508             if (is_nat_type(type)) {
11509                 return mk_lit(literal(*v));
11510             }
11511         }
11512         buffer<expr> args;
11513         expr fn = get_app_args(e, args);
11514         if (is_constant(fn)) {
11515             return visit_constant_core(fn, args, root);
11516         } else {
11517             fn = visit(fn, false);
11518             return visit_app_default(fn, args, root);
11519         }
11520     }
11521
11522     expr visit_proj(expr const &e, bool root) {
11523         expr v = visit(proj_expr(e), false);
11524         expr r = update_proj(e, v);
11525         return mk_let_decl(r, root);
11526     }
11527
11528     expr visit_mdata(expr const &e, bool root) {
11529         if (is_lc_mdata(e)) {
11530             expr v = visit(mdata_expr(e), false);
11531             expr r = mk_mdata(mdata_data(e), v);
11532             return mk_let_decl(r, root);
11533         } else {
11534             return visit(mdata_expr(e), root);
11535         }
11536     }
11537
11538     expr visit_lambda(expr e, bool root) {
11539         lean_assert(is_lambda(e));
11540         expr r;
11541         {
11542             flet<local_ctx> save_lctx(m_lctx, m_lctx);
11543             flet<cache> save_cache(m_cache, m_cache);
11544             unsigned old_fvars_size = m_fvars.size();
11545             buffer<expr> binding_fvars;
11546             while (is_lambda(e)) {
11547                 /* Types are ignored in compilation steps. So, we do not invoke
11548                  * visit for d. */
11549                 expr new_d =
```

```
11550                   instantiate_rev(binding_domain(e), binding_fvars.size(),
11551                                   binding_fvars.data());
11552               expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
11553                                                    new_d, binding_info(e));
11554               binding_fvars.push_back(new_fvar);
11555               e = binding_body(e);
11556           }
11557           expr new_body = visit(
11558               instantiate_rev(e, binding_fvars.size(), binding_fvars.data()),
11559               true);
11560           new_body = mk_let(old_fvars_size, new_body);
11561           r = m_lctx.mk_lambda(binding_fvars, new_body);
11562       }
11563       return mk_let_decl(r, root);
11564   }
11565
11566   expr visit_let(expr e, bool root) {
11567       buffer<expr> let_fvars;
11568       while (is_let(e)) {
11569           expr new_type = instantiate_rev(let_type(e), let_fvars.size(),
11570                                           let_fvars.data());
11571           bool val_as_root = is_lambda(let_value(e));
11572           expr new_val = visit(instantiate_rev(let_value(e), let_fvars.size(),
11573                                                let_fvars.data()),
11574                                val_as_root);
11575           name n = let_name(e);
11576           /* HACK:
11577              The `do` notation create "joinpoint". They are not real
11578              joinpoints since they may be nested in `HasBind.bind`
11579              applications. Moreover, the compiler currently inlines all local
11580              functions, and this creates a performance problem if we have a
11581              nontrivial number of "joinpoints" created by the `do` notation.
11582              The new compiler to be implemented in Lean itself will not use
11583              this naive inlining policy. In the meantime, we use the following
11584              HACK to control code size explosion. 1- We use
11585              `is_do_notation_joinpoint` to detect a joinpoint created by the
11586              `do` notation. 2- We encode them in the compiler as "pseudo
11587              joinpoints". 3- We disable inlining of "pseudo joinpoints" at
11588              `csimp`.
11589           */
11590           if (is_do_notation_joinpoint(n) ||
11591               should_create_let_decl(new_val, new_type)) {
11592               if (is_do_notation_joinpoint(n)) {
11593                   n = mk_pseudo_do_join_point_name(next_name());
11594               }
11595               expr new_fvar =
11596                   m_lctx.mk_local_decl(ngen(), n, new_type, new_val);
11597               let_fvars.push_back(new_fvar);
11598               m_fvars.push_back(new_fvar);
11599           } else {
11600               let_fvars.push_back(new_val);
11601           }
11602           e = let_body(e);
11603       }
11604       return visit(instantiate_rev(e, let_fvars.size(), let_fvars.data()),
11605                    root);
11606   }
11607
11608   bool has_never_extract(expr const &e) {
11609       expr const &fn = get_app_fn(e);
11610       return is_constant(fn) &&
11611              has_never_extract_attribute(env(), const_name(fn));
11612   }
11613
11614   expr cache_result(expr const &e, expr const &r, bool shared) {
11615       if (shared && !has_never_extract(e)) m_cache.insert(e, r);
11616       return r;
11617   }
11618
11619   expr visit(expr const &e, bool root) {
```

```
11620         switch (e.kind()) {
11621             case expr_kind::BVar:
11622             case expr_kind::MVar:
11623                 lean_unreachable();
11624             case expr_kind::FVar:
11625             case expr_kind::Sort:
11626             case expr_kind::Lit:
11627             case expr_kind::Pi:
11628                 return e;
11629             default:
11630                 break;
11631         }
11632
11633         if (is_lc_proof(e)) return e;
11634
11635         bool shared = is_shared(e);
11636         if (shared) {
11637             if (auto it = m_cache.find(e)) return *it;
11638         }
11639
11640         {
11641             type_checker tc(m_st, m_lctx);
11642             expr type = tc.whnf(tc.infer(e));
11643             if (is_sort(type)) {
11644                 /* Types are not pre-processed */
11645                 return cache_result(e, e, shared);
11646             } else if (tc.is_prop(type)) {
11647                 /* We replace proofs using `lc_proof` constant */
11648                 expr r = mk_app(mk_constant(get_lc_proof_name()), type);
11649                 return cache_result(e, r, shared);
11650             } else if (is_pi(type)) {
11651                 /* Functions that return types are not pre-processed. */
11652                 flet<local_ctx> save_lctx(m_lctx, m_lctx);
11653                 while (is_pi(type)) {
11654                     expr fvar = m_lctx.mk_local_decl(ngen(), binding_name(type),
11655                                                      binding_domain(type));
11656                     type = whnf(instantiate(binding_body(type), fvar));
11657                 }
11658                 if (is_sort(type)) return cache_result(e, e, shared);
11659             }
11660         }
11661
11662         switch (e.kind()) {
11663             case expr_kind::Const:
11664                 return cache_result(e, visit_constant(e, root), shared);
11665             case expr_kind::App:
11666                 return cache_result(e, visit_app(e, root), shared);
11667             case expr_kind::Proj:
11668                 return cache_result(e, visit_proj(e, root), shared);
11669             case expr_kind::MData:
11670                 return cache_result(e, visit_mdata(e, root), shared);
11671             case expr_kind::Lambda:
11672                 return cache_result(e, visit_lambda(e, root), shared);
11673             case expr_kind::Let:
11674                 return cache_result(e, visit_let(e, root), shared);
11675             default:
11676                 lean_unreachable();
11677         }
11678     }
11679
11680     expr operator()(expr const &e) {
11681         expr r = visit(e, true);
11682         return m_lctx.mk_lambda(m_fvars, r);
11683     }
11684 };
11685
11686 expr to_lcnf_core(environment const &env, local_ctx const &lctx,
11687                   expr const &e) {
11688     expr new_e = unfold_macro_defs(env, e);
11689     return to_lcnf_fn(env, lctx)(new_e);
```

```
11690 }
11691
11692 void initialize_lcnf() {}
11693
11694 void finalize_lcnf() {}
11695 }  // namespace lean
11696 // ::::::::::::::
11697 // compiler/ll_infer_type.cpp
11698 // ::::::::::::::
11699 /*
11700 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
11701 Released under Apache 2.0 license as described in the file LICENSE.
11702
11703 Author: Leonardo de Moura
11704 */
11705 #include <lean/flet.h>
11706 #include <lean/sstream.h>
11707
11708 #include "kernel/instantiate.h"
11709 #include "kernel/replace_fn.h"
11710 #include "library/compiler/extern_attribute.h"
11711 #include "library/compiler/util.h"
11712
11713 namespace lean {
11714 static expr *g_bot = nullptr;
11715
11716 /* Infer type of expressions in ENF or LLNF. */
11717 class ll_infer_type_fn {
11718     type_checker::state m_st;
11719     local_ctx m_lctx;
11720     buffer<name> const *m_new_decl_names{nullptr};
11721     buffer<expr> const *m_new_decl_types{nullptr};
11722
11723     environment const &env() const { return m_st.env(); }
11724     name_generator &ngen() { return m_st.ngen(); }
11725
11726     bool may_use_bot() const { return m_new_decl_types != nullptr; }
11727
11728     expr infer_lambda(expr e) {
11729         flet<local_ctx> save_lctx(m_lctx, m_lctx);
11730         buffer<expr> fvars;
11731         while (is_lambda(e)) {
11732             lean_assert(!has_loose_bvars(binding_domain(e)));
11733             expr fvar = m_lctx.mk_local_decl(ngen(), binding_name(e),
11734                                              binding_domain(e));
11735             fvars.push_back(fvar);
11736             e = binding_body(e);
11737         }
11738         expr r = infer(instantiate_rev(e, fvars.size(), fvars.data()));
11739         return m_lctx.mk_pi(fvars, r);
11740     }
11741
11742     expr infer_let(expr e) {
11743         flet<local_ctx> save_lctx(m_lctx, m_lctx);
11744         buffer<expr> fvars;
11745         while (is_let(e)) {
11746             lean_assert(!has_loose_bvars(let_type(e)));
11747             expr type;
11748             if (is_join_point_name(let_name(e))) {
11749                 expr val =
11750                     instantiate_rev(let_value(e), fvars.size(), fvars.data());
11751                 type = infer(val);
11752             } else {
11753                 type = let_type(e);
11754             }
11755             expr fvar = m_lctx.mk_local_decl(ngen(), let_name(e), type);
11756             fvars.push_back(fvar);
11757             e = let_body(e);
11758         }
11759         return infer(instantiate_rev(e, fvars.size(), fvars.data()));
```

```
11760     }
11761
11762     expr infer_cases(expr const &e) {
11763         buffer<expr> args;
11764         get_app_args(e, args);
11765         lean_assert(args.size() >= 2);
11766         bool first = true;
11767         expr r = *g_bot;
11768         for (unsigned i = 1; i < args.size(); i++) {
11769             expr minor = args[i];
11770             buffer<expr> fvars;
11771             while (is_lambda(minor)) {
11772                 lean_assert(!has_loose_bvars(binding_domain(minor)));
11773                 expr fvar = m_lctx.mk_local_decl(ngen(), binding_name(minor),
11774                                                  binding_domain(minor));
11775                 fvars.push_back(fvar);
11776                 minor = binding_body(minor);
11777             }
11778             expr minor_type =
11779                 infer(instantiate_rev(minor, fvars.size(), fvars.data()));
11780             if (minor_type == mk_enf_object_type()) {
11781                 /* If one of the branches return `_obj`, then the resultant type
11782                    is `_obj`,
11783                    and the other branches should box result if it is not `_obj`.
11784                  */
11785                 return minor_type;
11786             } else if (minor_type == *g_bot) {
11787                 /* Ignore*/
11788             } else if (first) {
11789                 r = minor_type;
11790                 first = false;
11791             } else if (minor_type != r) {
11792                 /* All branches should return the same type, otherwise we box
11793                  * them. */
11794                 return mk_enf_object_type();
11795             }
11796         }
11797         lean_assert(may_use_bot() || r != *g_bot);
11798         return r;
11799     }
11800
11801     expr infer_constructor_type(expr const &e) {
11802         name I_name = env()
11803                           .get(const_name(get_app_fn(e)))
11804                           .to_constructor_val()
11805                           .get_induct();
11806         if (optional<unsigned> sz = ::lean::is_enum_type(env(), I_name)) {
11807             if (optional<expr> uint = to_uint_type(*sz)) return *uint;
11808         }
11809         return mk_enf_object_type();
11810     }
11811
11812     expr infer_app(expr const &e) {
11813         if (is_cases_on_app(env(), e)) {
11814             return infer_cases(e);
11815         } else if (is_constructor_app(env(), e)) {
11816             return infer_constructor_type(e);
11817         } else if (is_app_of(e, get_panic_name())) {
11818             /*
11819                We should treat `panic` as `unreachable`.
11820                Otherwise, we will not infer the correct type IRType for
11821                ```
11822                def f (n : UInt32) : UInt32 :=
11823                if n == 0 then panic! "foo"
11824                else n+1
11825                ```
11826                Reason: `panic! "foo"` is expanded into
11827                ```
11828                let _x_1 : String := mkPanicMessage "<file-name>" 2 15 "foo" in
11829                @panic.{0} UInt32 UInt32.Inhabited _x_1
```

```
11830               ```
11831               and `panic` can't be specialize because it is a primitive
11832               implemented in C++, and if we don't do anything it will assume
11833               `panic` returns an `_obj`.
11834            */
11835           return may_use_bot() ? *g_bot : mk_enf_object_type();
11836       } else {
11837           expr const &fn = get_app_fn(e);
11838           expr fn_type = infer(fn);
11839           lean_assert(may_use_bot() || fn_type != *g_bot);
11840           if (fn_type == *g_bot) return *g_bot;
11841           unsigned nargs = get_app_num_args(e);
11842           for (unsigned i = 0; i < nargs; i++) {
11843               if (!is_pi(fn_type)) {
11844                   return mk_enf_object_type();
11845               } else {
11846                   fn_type = binding_body(fn_type);
11847                   lean_assert(!has_loose_bvars(fn_type));
11848               }
11849           }
11850           if (is_pi(fn_type)) {
11851               /* Application is creating a closure. */
11852               return mk_enf_object_type();
11853           } else {
11854               return fn_type;
11855           }
11856       }
11857   }
11858
11859   optional<unsigned> is_enum_type(expr const &type) {
11860       expr const &I = get_app_fn(type);
11861       if (!is_constant(I)) return optional<unsigned>();
11862       return ::lean::is_enum_type(env(), const_name(I));
11863   }
11864
11865   expr infer_proj(expr const &e) {
11866       name const &I_name = proj_sname(e);
11867       inductive_val I_val = env().get(I_name).to_inductive_val();
11868       lean_assert(I_val.get_ncnstrs() == 1);
11869       name const &k_name = head(I_val.get_cnstrs());
11870       constant_info k_info = env().get(k_name);
11871       expr type = k_info.get_type();
11872       unsigned nparams = I_val.get_nparams();
11873       buffer<expr> telescope;
11874       local_ctx lctx;
11875       to_telescope(env(), lctx, ngen(), type, telescope);
11876       lean_assert(telescope.size() >= nparams);
11877       lean_assert(nparams + proj_idx(e).get_small_value() < telescope.size());
11878       type_checker tc(m_st, lctx);
11879       expr ftype =
11880           lctx.get_type(telescope[nparams + proj_idx(e).get_small_value()]);
11881       ftype = tc.whnf(ftype);
11882       if (is_constant(ftype) && is_runtime_scalar_type(const_name(ftype))) {
11883           return ftype;
11884       } else if (optional<unsigned> sz = is_enum_type(ftype)) {
11885           if (optional<expr> uint = to_uint_type(*sz)) return *uint;
11886       }
11887       return mk_enf_object_type();
11888   }
11889
11890   expr infer_constant(expr const &e) {
11891       if (optional<expr> type =
11892               get_extern_constant_ll_type(env(), const_name(e))) {
11893           return *type;
11894       } else if (is_constructor(env(), const_name(e))) {
11895           return infer_constructor_type(e);
11896       } else if (is_enf_neutral(e)) {
11897           return mk_enf_neutral_type();
11898       } else if (is_enf_unreachable(e)) {
11899           return may_use_bot() ? *g_bot : mk_enf_object_type();
```

```
11900         } else {
11901             name c = mk_cstage2_name(const_name(e));
11902             optional<constant_info> info = env().find(c);
11903             if (info) return info->get_type();
11904             if (m_new_decl_types) {
11905                 lean_assert(m_new_decl_names->size() ==
11906                             m_new_decl_types->size());
11907                 for (unsigned i = 0; i < m_new_decl_names->size(); i++) {
11908                     if (const_name(e) == (*m_new_decl_names)[i])
11909                         return (*m_new_decl_types)[i];
11910                 }
11911                 return *g_bot;
11912             }
11913             throw exception(sstream() << "compiler failed to infer low level "
11914                                       << "type, unknown declaration '"
11915                                    << const_name(e) << "'");
11916         }
11917     }
11918
11919     expr infer(expr const &e) {
11920         switch (e.kind()) {
11921             case expr_kind::App:
11922                 return infer_app(e);
11923             case expr_kind::Lambda:
11924                 return infer_lambda(e);
11925             case expr_kind::Let:
11926                 return infer_let(e);
11927             case expr_kind::Proj:
11928                 return infer_proj(e);
11929             case expr_kind::Const:
11930                 return infer_constant(e);
11931             case expr_kind::MData:
11932                 return infer(mdata_expr(e));
11933             case expr_kind::Lit:
11934                 return mk_enf_object_type();
11935             case expr_kind::FVar:
11936                 return m_lctx.get_local_decl(e).get_type();
11937             case expr_kind::Sort:
11938                 return mk_enf_neutral_type();
11939             case expr_kind::Pi:
11940                 return mk_enf_neutral_type();
11941             case expr_kind::BVar:
11942                 lean_unreachable();
11943             case expr_kind::MVar:
11944                 lean_unreachable();
11945         }
11946         lean_unreachable();
11947     }
11948
11949    public:
11950     ll_infer_type_fn(environment const &env, buffer<name> const &ns,
11951                      buffer<expr> const &ts)
11952         : m_st(env), m_new_decl_names(&ns), m_new_decl_types(&ts) {}
11953     ll_infer_type_fn(environment const &env, local_ctx const &lctx)
11954         : m_st(env), m_lctx(lctx) {}
11955     expr operator()(expr const &e) { return infer(e); }
11956 };
11957
11958 void ll_infer_type(environment const &env, comp_decls const &ds,
11959                    buffer<expr> &ts) {
11960     buffer<name> ns;
11961     ts.clear();
11962     /* Initialize `ts` */
11963     for (comp_decl const &d : ds) {
11964         /* For mutually recursive declarations `t` may contain `_bot`. */
11965         expr t = ll_infer_type_fn(env, ns, ts)(d.snd());
11966         ns.push_back(d.fst());
11967         ts.push_back(t);
11968     }
11969     /* Keep refining types in `ts` until fix point */
```

```
11970     while (true) {
11971         bool modified = false;
11972         unsigned i = 0;
11973         for (comp_decl const &d : ds) {
11974             expr t1 = ll_infer_type_fn(env, ns, ts)(d.snd());
11975             if (t1 != ts[i]) {
11976                 modified = true;
11977                 ts[i] = t1;
11978             }
11979             i++;
11980         }
11981         if (!modified) break;
11982     }
11983     /* `ts` may still contain `_bot` for non-terminating or bogus programs.
11984        Example: `def f (x) := f (f x)`.
11985
11986        It is safe to replace `_bot` with `_obj`. */
11987     for (expr &t : ts) {
11988         t = replace(t, [&](expr const &e, unsigned) {
11989             if (e == *g_bot)
11990                 return some_expr(mk_enf_object_type());
11991             else
11992                 return none_expr();
11993         });
11994     }
11995     lean_assert(ts.size() == length(ds));
11996 }
11997
11998 expr ll_infer_type(environment const &env, local_ctx const &lctx,
11999                    expr const &e) {
12000     return ll_infer_type_fn(env, lctx)(e);
12001 }
12002
12003 void initialize_ll_infer_type() {
12004     g_bot = new expr(mk_constant("_bot"));
12005     mark_persistent(g_bot->raw());
12006 }
12007
12008 void finalize_ll_infer_type() { delete g_bot; }
12009 }  // namespace lean
12010 // ::::::::::::::
12011 // compiler/llnf.cpp
12012 // ::::::::::::::
12013 /*
12014 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
12015 Released under Apache 2.0 license as described in the file LICENSE.
12016
12017 Author: Leonardo de Moura
12018 */
12019 #include <lean/flet.h>
12020 #include <lean/sstream.h>
12021
12022 #include <algorithm>
12023 #include <limits>
12024 #include <unordered_set>
12025 #include <vector>
12026
12027 #include "kernel/abstract.h"
12028 #include "kernel/for_each_fn.h"
12029 #include "kernel/instantiate.h"
12030 #include "library/compiler/borrowed_annotation.h"
12031 #include "library/compiler/cse.h"
12032 #include "library/compiler/elim_dead_let.h"
12033 #include "library/compiler/extern_attribute.h"
12034 #include "library/compiler/ir.h"
12035 #include "library/compiler/ll_infer_type.h"
12036 #include "library/compiler/llnf.h"
12037 #include "library/compiler/util.h"
12038 #include "library/trace.h"
12039 #include "library/util.h"
```

```
12040 #include "util/name_hash_map.h"
12041 #include "util/name_hash_set.h"
12042
12043 namespace lean {
12044 static expr *g_apply = nullptr;
12045 static expr *g_closure = nullptr;
12046 static char const *g_cnstr = "_cnstr";
12047 static name *g_reuse = nullptr;
12048 static name *g_reset = nullptr;
12049 static name *g_fset = nullptr;
12050 static name *g_sset = nullptr;
12051 static name *g_uset = nullptr;
12052 static name *g_proj = nullptr;
12053 static name *g_sproj = nullptr;
12054 static name *g_fproj = nullptr;
12055 static name *g_uproj = nullptr;
12056 static expr *g_jmp = nullptr;
12057 static name *g_box = nullptr;
12058 static name *g_unbox = nullptr;
12059 static expr *g_inc = nullptr;
12060 static expr *g_dec = nullptr;
12061
12062 expr mk_llnf_apply() { return *g_apply; }
12063 bool is_llnf_apply(expr const &e) { return e == *g_apply; }
12064
12065 expr mk_llnf_closure() { return *g_closure; }
12066 bool is_llnf_closure(expr const &e) { return e == *g_closure; }
12067
12068 static bool is_llnf_unary_primitive(expr const &e, name const &prefix,
12069                                     unsigned &i) {
12070     if (!is_constant(e)) return false;
12071     name const &n = const_name(e);
12072     if (!is_internal_name(n) || n.is_atomic() || !n.is_numeral() ||
12073         n.get_prefix() != prefix)
12074         return false;
12075     i = n.get_numeral().get_small_value();
12076     return true;
12077 }
12078
12079 static bool is_llnf_binary_primitive(expr const &e, name const &prefix,
12080                                      unsigned &i1, unsigned &i2) {
12081     if (!is_constant(e)) return false;
12082     name const &n2 = const_name(e);
12083     if (n2.is_atomic() || !n2.is_numeral()) return false;
12084     i2 = n2.get_numeral().get_small_value();
12085     name const &n1 = n2.get_prefix();
12086     if (n1.is_atomic() || !n1.is_numeral() || n1.get_prefix() != prefix)
12087         return false;
12088     i1 = n1.get_numeral().get_small_value();
12089     return true;
12090 }
12091
12092 static bool is_llnf_ternary_primitive(expr const &e, name const &prefix,
12093                                       unsigned &i1, unsigned &i2,
12094                                       unsigned &i3) {
12095     if (!is_constant(e)) return false;
12096     name const &n3 = const_name(e);
12097     if (!is_internal_name(n3)) return false;
12098     if (n3.is_atomic() || !n3.is_numeral()) return false;
12099     i3 = n3.get_numeral().get_small_value();
12100     name const &n2 = n3.get_prefix();
12101     if (n2.is_atomic() || !n2.is_numeral()) return false;
12102     i2 = n2.get_numeral().get_small_value();
12103     name const &n1 = n2.get_prefix();
12104     if (n1.is_atomic() || !n1.is_numeral() || n1.get_prefix() != prefix)
12105         return false;
12106     i1 = n1.get_numeral().get_small_value();
12107     return true;
12108 }
12109
```

```
12110 static bool is_llnf_quaternary_primitive(expr const &e, name const &prefix,
12111                                           unsigned &i1, unsigned &i2,
12112                                           unsigned &i3, unsigned &i4) {
12113     if (!is_constant(e)) return false;
12114     name const &n4 = const_name(e);
12115     if (!is_internal_name(n4)) return false;
12116     if (n4.is_atomic() || !n4.is_numeral()) return false;
12117     i4 = n4.get_numeral().get_small_value();
12118     name const &n3 = n4.get_prefix();
12119     if (!is_internal_name(n3)) return false;
12120     if (n3.is_atomic() || !n3.is_numeral()) return false;
12121     i3 = n3.get_numeral().get_small_value();
12122     name const &n2 = n3.get_prefix();
12123     if (n2.is_atomic() || !n2.is_numeral()) return false;
12124     i2 = n2.get_numeral().get_small_value();
12125     name const &n1 = n2.get_prefix();
12126     if (n1.is_atomic() || !n1.is_numeral() || n1.get_prefix() != prefix)
12127         return false;
12128     i1 = n1.get_numeral().get_small_value();
12129     return true;
12130 }
12131
12132 /*
12133 A constructor object contains a header, then a sequence of pointers to other
12134 Lean objects, a sequence of `usize` (i.e., `size_t`) scalar values, and a
12135 sequence of other scalar values. We store pointer and `usize` objects before
12136 other scalar values to simplify how we compute the position where data is
12137 stored. For example, the "instruction" `_sproj.4.2.3 o` access a value of size 4
12138 at offset `sizeof(void*)*2+3`. We have considered a simpler representation where
12139 we just have the size and offset, we decided to not used it because we would
12140 have to generate different C++ code for 32 and 64 bit machines. This would
12141 complicate the bootstrapping process. We store the `usize` scalar values before
12142 other scalar values because their size is platform specific. We also have custom
12143 instructions (`_uset` and `_uproj`) to set and retrieve `usize` scalar fields.
12144 */
12145
12146 /* The `I._cnstr.<cidx>.<num_usizes>.<num_bytes>` instruction constructs a
12147  * constructor object with tag `cidx`, and scalar area with space for
12148  * `num_usize` `usize` values + `num_bytes` bytes. */
12149 expr mk_llnf_cnstr(name const &I, unsigned cidx, unsigned num_usizes,
12150                    unsigned num_bytes) {
12151     return mk_constant(
12152         name(name(name(name(I, g_cnstr), cidx), num_usizes), num_bytes));
12153 }
12154 bool is_llnf_cnstr(expr const &e, name &I, unsigned &cidx, unsigned &num_usizes,
12155                    unsigned &num_bytes) {
12156     if (!is_constant(e)) return false;
12157     name const &n3 = const_name(e);
12158     if (!is_internal_name(n3)) return false;
12159     if (n3.is_atomic() || !n3.is_numeral()) return false;
12160     num_bytes = n3.get_numeral().get_small_value();
12161     name const &n2 = n3.get_prefix();
12162     if (n2.is_atomic() || !n2.is_numeral()) return false;
12163     num_usizes = n2.get_numeral().get_small_value();
12164     name const &n1 = n2.get_prefix();
12165     if (n1.is_atomic() || !n1.is_numeral()) return false;
12166     cidx = n1.get_numeral().get_small_value();
12167     name const &n0 = n1.get_prefix();
12168     if (n0.is_atomic() || !n0.is_string() || n0.get_string() != g_cnstr)
12169         return false;
12170     I = n0.get_prefix();
12171     return true;
12172 }
12173
12174 /* The `_reuse.<cidx>.<num_usizes>.<num_bytes>.<updt_cidx>` is similar to
12175  * `_cnstr.<cidx>.<num_usize>.<num_bytes>`, but it takes an extra argument: a
12176  * memory cell that may be reused. */
12177 expr mk_llnf_reuse(unsigned cidx, unsigned num_usizes, unsigned num_bytes,
12178                    bool updt_cidx) {
12179     return mk_constant(name(
```

```
12180         name(name(name(*g_reuse, cidx), num_usizes), num_bytes), updt_cidx));
12181 }
12182 bool is_llnf_reuse(expr const &e, unsigned &cidx, unsigned &num_usizes,
12183                    unsigned &num_bytes, bool &updt_cidx) {
12184     unsigned aux = 0;
12185     bool r = is_llnf_quaternary_primitive(e, *g_reuse, cidx, num_usizes,
12186                                           num_bytes, aux);
12187     updt_cidx = aux;
12188     return r;
12189 }
12190
12191 expr mk_llnf_reset(unsigned n) { return mk_constant(name(*g_reset, n)); }
12192 bool is_llnf_reset(expr const &e, unsigned &n) {
12193     return is_llnf_unary_primitive(e, *g_reset, n);
12194 }
12195
12196 /* The `_sset.<sz>.<n>.<offset>` instruction sets a scalar value of size `sz`
12197  * (in bytes) at offset `sizeof(void*)*n + offset`. The value `n` is the number
12198  * of pointer and `usize` fields. */
12199 expr mk_llnf_sset(unsigned sz, unsigned n, unsigned offset) {
12200     return mk_constant(name(name(name(*g_sset, sz), n), offset));
12201 }
12202 bool is_llnf_sset(expr const &e, unsigned &sz, unsigned &n, unsigned &offset) {
12203     return is_llnf_ternary_primitive(e, *g_sset, sz, n, offset);
12204 }
12205
12206 expr mk_llnf_fset(unsigned n, unsigned offset) {
12207     return mk_constant(name(name(*g_fset, n), offset));
12208 }
12209 bool is_llnf_fset(expr const &e, unsigned &n, unsigned &offset) {
12210     return is_llnf_binary_primitive(e, *g_fset, n, offset);
12211 }
12212
12213 /* The `_uset.<n>` instruction sets a `usize` value in a constructor object at
12214  * offset `sizeof(void*)*n`. */
12215 expr mk_llnf_uset(unsigned n) { return mk_constant(name(*g_uset, n)); }
12216 bool is_llnf_uset(expr const &e, unsigned &n) {
12217     return is_llnf_unary_primitive(e, *g_uset, n);
12218 }
12219
12220 /* The `_proj.<idx>` instruction retrieves an object field in a constructor
12221  * object at offset `sizeof(void*)*idx` */
12222 expr mk_llnf_proj(unsigned idx) { return mk_constant(name(*g_proj, idx)); }
12223 bool is_llnf_proj(expr const &e, unsigned &idx) {
12224     return is_llnf_unary_primitive(e, *g_proj, idx);
12225 }
12226
12227 /* The `_sproj.<sz>.<n>.<offset>` instruction retrieves a scalar field of size
12228  * `sz` (in bytes) in a constructor object at offset `sizeof(void*)*n + offset`.
12229  * The value `n` is the number of pointer and `usize` fields. */
12230 expr mk_llnf_sproj(unsigned sz, unsigned n, unsigned offset) {
12231     return mk_constant(name(name(name(*g_sproj, sz), n), offset));
12232 }
12233 bool is_llnf_sproj(expr const &e, unsigned &sz, unsigned &n, unsigned &offset) {
12234     return is_llnf_ternary_primitive(e, *g_sproj, sz, n, offset);
12235 }
12236
12237 expr mk_llnf_fproj(unsigned n, unsigned offset) {
12238     return mk_constant(name(name(*g_sproj, n), offset));
12239 }
12240 bool is_llnf_fproj(expr const &e, unsigned &n, unsigned &offset) {
12241     return is_llnf_binary_primitive(e, *g_fproj, n, offset);
12242 }
12243
12244 /* The `_uproj.<idx>` instruction retrieves an `usize` field in a constructor
12245  * object at offset `sizeof(void*)*idx` */
12246 expr mk_llnf_uproj(unsigned idx) { return mk_constant(name(*g_uproj, idx)); }
12247 bool is_llnf_uproj(expr const &e, unsigned &idx) {
12248     return is_llnf_unary_primitive(e, *g_uproj, idx);
12249 }
```

```
12250
12251 /* The `_jmp` instruction is a "jump" to a join point. */
12252 expr mk_llnf_jmp() { return *g_jmp; }
12253 bool is_llnf_jmp(expr const &e) { return e == *g_jmp; }
12254
12255 /* The `_box.<n>` instruction converts an unboxed value (type `uint*`) into a
12256    boxed value (type `_obj`). The parameter `n` specifies the number of bytes
12257    necessary to store the unboxed value. This information could be also
12258    retrieved from the type of the variable being boxed, but for simplicity, we
12259    store it in the instruction too.
12260
12261    Remark: we use the instruction `_box.0` to box unboxed values of type `usize`
12262    into a boxed value (type `_obj`). We use `0` because the number of bytes
12263    necessary to store a `usize` is different in 32 and 64 bit machines. */
12264 expr mk_llnf_box(unsigned n) { return mk_constant(name(*g_box, n)); }
12265 bool is_llnf_box(expr const &e, unsigned &n) {
12266     return is_llnf_unary_primitive(e, *g_box, n);
12267 }
12268
12269 /* The `_unbox.<n>` instruction converts a boxed value (type `_obj`) into an
12270    unboxed value (type `uint*` or `usize`). The parameter `n` specifies the
12271    number of bytes necessary to store the unboxed value. It is not really
12272    needed, but we use to keep it consistent with `_box.<n>`.
12273
12274    Remark: we use the instruction `_unbox.0` like we use `_box.0`. */
12275 expr mk_llnf_unbox(unsigned n) { return mk_constant(name(*g_unbox, n)); }
12276 bool is_llnf_unbox(expr const &e, unsigned &n) {
12277     return is_llnf_unary_primitive(e, *g_unbox, n);
12278 }
12279
12280 expr mk_llnf_inc() { return *g_inc; }
12281 bool is_llnf_inc(expr const &e) { return e == *g_inc; }
12282
12283 expr mk_llnf_dec() { return *g_dec; }
12284 bool is_llnf_dec(expr const &e) { return e == *g_dec; }
12285
12286 bool is_llnf_op(expr const &e) {
12287     return is_llnf_closure(e) || is_llnf_apply(e) || is_llnf_cnstr(e) ||
12288            is_llnf_reuse(e) || is_llnf_reset(e) || is_llnf_sset(e) ||
12289            is_llnf_fset(e) || is_llnf_uset(e) || is_llnf_proj(e) ||
12290            is_llnf_sproj(e) || is_llnf_fproj(e) || is_llnf_uproj(e) ||
12291            is_llnf_jmp(e) || is_llnf_box(e) || is_llnf_unbox(e) ||
12292            is_llnf_inc(e) || is_llnf_dec(e);
12293 }
12294
12295 cnstr_info::cnstr_info(unsigned cidx, list<field_info> const &finfo)
12296     : m_cidx(cidx), m_field_info(finfo) {
12297     for (field_info const &info : finfo) {
12298         if (info.m_kind == field_info::Object)
12299             m_num_objs++;
12300         else if (info.m_kind == field_info::USize)
12301             m_num_usizes++;
12302         else if (info.m_kind == field_info::Scalar)
12303             m_scalar_sz += info.m_size;
12304     }
12305 }
12306
12307 unsigned get_llnf_arity(environment const &env, name const &n) {
12308     /* First, try to infer arity from `_cstage2` auxiliary definition. */
12309     name c = mk_cstage2_name(n);
12310     optional<constant_info> info = env.find(c);
12311     if (info && info->is_definition()) {
12312         return get_num_nested_lambdas(info->get_value());
12313     }
12314     optional<unsigned> arity = get_extern_constant_arity(env, n);
12315     if (!arity)
12316         throw exception(sstream()
12317                         << "code generation failed, unknown '" << n << "'");
12318     return *arity;
12319 }
```

```
12320
12321 static void get_cnstr_info_core(type_checker::state &st, name const &n,
12322                                 buffer<field_info> &result) {
12323     environment const &env = st.env();
12324     constant_info info = env.get(n);
12325     lean_assert(info.is_constructor());
12326     constructor_val val = info.to_constructor_val();
12327     expr type = info.get_type();
12328     name I_name = val.get_induct();
12329     unsigned nparams = val.get_nparams();
12330     local_ctx lctx;
12331     buffer<expr> telescope;
12332     unsigned next_object = 0;
12333     unsigned max_scalar_size = 0;
12334     to_telescope(env, lctx, st.ngen(), type, telescope);
12335     lean_assert(telescope.size() >= nparams);
12336     for (unsigned i = nparams; i < telescope.size(); i++) {
12337         expr ftype = lctx.get_type(telescope[i]);
12338         if (is_irrelevant_type(st, lctx, ftype)) {
12339             result.push_back(field_info::mk_irrelevant());
12340         } else {
12341             type_checker tc(st, lctx);
12342             ftype = tc.whnf(ftype);
12343             if (is_usize_type(ftype)) {
12344                 result.push_back(field_info::mk_usize());
12345             } else if (optional<unsigned> sz = is_builtin_scalar(ftype)) {
12346                 max_scalar_size = std::max(*sz, max_scalar_size);
12347                 result.push_back(field_info::mk_scalar(*sz, ftype));
12348             } else if (optional<unsigned> sz = is_enum_type(env, ftype)) {
12349                 optional<expr> uint = to_uint_type(*sz);
12350                 if (!uint)
12351                     throw exception(
12352                         "code generation failed, enumeration type is too big");
12353                 max_scalar_size = std::max(*sz, max_scalar_size);
12354                 result.push_back(field_info::mk_scalar(*sz, *uint));
12355             } else {
12356                 result.push_back(field_info::mk_object(next_object));
12357                 next_object++;
12358             }
12359         }
12360     }
12361
12362     unsigned next_idx = next_object;
12363     /* Remark:
12364        - usize fields are stored after object fields.
12365        - regular scalar fields are stored after object and usize fields,
12366          and are sorted by size. */
12367     /* Fix USize idxs */
12368     for (field_info &info : result) {
12369         if (info.m_kind == field_info::USize) {
12370             info.m_idx = next_idx;
12371             next_idx++;
12372         }
12373     }
12374     unsigned idx = next_idx;
12375     unsigned offset = 0;
12376     /* Fix regular scalar offsets and idxs */
12377     for (unsigned sz = max_scalar_size; sz > 0; sz--) {
12378         for (field_info &info : result) {
12379             if (info.m_kind == field_info::Scalar && info.m_size == sz) {
12380                 info.m_idx = idx;
12381                 info.m_offset = offset;
12382                 offset += info.m_size;
12383             }
12384         }
12385     }
12386 }
12387
12388 cnstr_info get_cnstr_info(type_checker::state &st, name const &n) {
12389     buffer<field_info> finfos;
```

```
12390     get_cnstr_info_core(st, n, finfos);
12391     unsigned cidx = get_constructor_idx(st.env(), n);
12392     return cnstr_info(cidx, to_list(finfos));
12393 }
12394
12395 class to_lambda_pure_fn {
12396     typedef name_hash_set name_set;
12397     typedef name_hash_map<cnstr_info> cnstr_info_cache;
12398     typedef name_hash_map<optional<unsigned>> enum_cache;
12399     type_checker::state m_st;
12400     local_ctx m_lctx;
12401     buffer<expr> m_fvars;
12402     name m_x;
12403     name m_j;
12404     unsigned m_next_idx{1};
12405     unsigned m_next_jp_idx{1};
12406     cnstr_info_cache m_cnstr_info_cache;
12407
12408     environment const &env() const { return m_st.env(); }
12409
12410     name_generator &ngen() { return m_st.ngen(); }
12411
12412     optional<unsigned> is_enum_type(expr const &type) {
12413         return ::lean::is_enum_type(env(), type);
12414     }
12415
12416     unsigned get_arity(name const &n) const {
12417         return ::lean::get_llnf_arity(env(), n);
12418     }
12419
12420     bool is_join_point_app(expr const &e) {
12421         expr const &fn = get_app_fn(e);
12422         if (!is_fvar(fn)) return false;
12423         local_decl d = m_lctx.get_local_decl(fn);
12424         return is_join_point_name(d.get_user_name());
12425     }
12426
12427     expr ensure_terminal(expr const &e) {
12428         lean_assert(!is_let(e) && !is_lambda(e));
12429         if (is_cases_on_app(env(), e) || is_fvar(e) || is_join_point_app(e) ||
12430             is_enf_unreachable(e)) {
12431             return e;
12432         } else {
12433             expr type = ll_infer_type(env(), m_lctx, e);
12434             if (is_pi(type)) {
12435                 /* It is a closure. */
12436                 type = mk_enf_object_type();
12437             }
12438             return ::lean::mk_let("_res", type, e, mk_bvar(0));
12439         }
12440     }
12441
12442     expr mk_llnf_app(expr const &fn, buffer<expr> const &args) {
12443         lean_assert(is_fvar(fn) || is_constant(fn));
12444         if (is_fvar(fn)) {
12445             local_decl d = m_lctx.get_local_decl(fn);
12446             if (is_join_point_name(d.get_user_name())) {
12447                 return mk_app(mk_app(mk_llnf_jmp(), fn), args);
12448             } else {
12449                 return mk_app(mk_app(mk_llnf_apply(), fn), args);
12450             }
12451         } else {
12452             lean_assert(is_constant(fn));
12453             if (is_enf_neutral(fn)) {
12454                 return mk_enf_neutral();
12455             } else if (is_enf_unreachable(fn)) {
12456                 return mk_enf_unreachable();
12457             } else {
12458                 unsigned arity = get_arity(const_name(fn));
12459                 if (args.size() == arity) {
```

```
12460                     return mk_app(fn, args);
12461                 } else if (args.size() < arity) {
12462                     /* Under application: create closure. */
12463                     return mk_app(mk_app(mk_llnf_closure(), fn), args);
12464                 } else {
12465                     /* Over application. */
12466                     lean_assert(args.size() > arity);
12467                     expr new_fn = m_lctx.mk_local_decl(
12468                         ngen(), next_name(), mk_enf_object_type(),
12469                         mk_app(fn, arity, args.data()));
12470                     m_fvars.push_back(new_fn);
12471                     return mk_app(mk_app(mk_llnf_apply(), new_fn),
12472                                   args.size() - arity, args.data() + arity);
12473                 }
12474             }
12475         }
12476     }
12477
12478     cnstr_info get_cnstr_info(name const &n) {
12479         auto it = m_cnstr_info_cache.find(n);
12480         if (it != m_cnstr_info_cache.end()) return it->second;
12481         cnstr_info r = ::lean::get_cnstr_info(m_st, n);
12482         m_cnstr_info_cache.insert(mk_pair(n, r));
12483         return r;
12484     }
12485
12486     name next_name() {
12487         name r = m_x.append_after(m_next_idx);
12488         m_next_idx++;
12489         return r;
12490     }
12491
12492     name next_jp_name() {
12493         name r = m_j.append_after(m_next_jp_idx);
12494         m_next_jp_idx++;
12495         return mk_join_point_name(r);
12496     }
12497
12498     expr mk_let(unsigned saved_fvars_size, expr r) {
12499         lean_assert(saved_fvars_size <= m_fvars.size());
12500         if (saved_fvars_size == m_fvars.size()) return r;
12501         buffer<expr> used;
12502         name_hash_set used_fvars;
12503         collect_used(r, used_fvars);
12504         while (m_fvars.size() > saved_fvars_size) {
12505             expr x = m_fvars.back();
12506             m_fvars.pop_back();
12507             if (used_fvars.find(fvar_name(x)) == used_fvars.end()) {
12508                 continue;
12509             }
12510             local_decl x_decl = m_lctx.get_local_decl(x);
12511             expr val = *x_decl.get_value();
12512             collect_used(val, used_fvars);
12513             used.push_back(x);
12514         }
12515         std::reverse(used.begin(), used.end());
12516         return m_lctx.mk_lambda(used, r);
12517     }
12518
12519     expr visit_let(expr e) {
12520         buffer<expr> fvars;
12521         while (is_let(e)) {
12522             lean_assert(!has_loose_bvars(let_type(e)));
12523             expr new_val = visit(
12524                 instantiate_rev(let_value(e), fvars.size(), fvars.data()));
12525             name n = let_name(e);
12526             if (is_internal_name(n)) {
12527                 if (is_join_point_name(n))
12528                     n = next_jp_name();
12529                 else
```

```
12530                   n = next_name();
12531           }
12532           expr new_type = let_type(e);
12533           if (is_llnf_proj(get_app_fn(new_val))) {
12534               /* Ensure new_type is `_obj`. This is important for polymorphic
12535                  types instantiated with scalar values (e.g., `prod bool
12536                  bool`). */
12537               new_type = mk_enf_object_type();
12538           }
12539           expr new_fvar = m_lctx.mk_local_decl(ngen(), n, new_type, new_val);
12540           fvars.push_back(new_fvar);
12541           m_fvars.push_back(new_fvar);
12542           e = let_body(e);
12543       }
12544       e = instantiate_rev(e, fvars.size(), fvars.data());
12545       lean_assert(!is_let(e));
12546       e = ensure_terminal(e);
12547       return visit(e);
12548   }
12549
12550   expr visit_lambda(expr e) {
12551       buffer<expr> binding_fvars;
12552       while (is_lambda(e)) {
12553           lean_assert(!has_loose_bvars(binding_domain(e)));
12554           expr new_fvar = m_lctx.mk_local_decl(
12555               ngen(), next_name(), binding_domain(e), binding_info(e));
12556           binding_fvars.push_back(new_fvar);
12557           e = binding_body(e);
12558       }
12559       e = instantiate_rev(e, binding_fvars.size(), binding_fvars.data());
12560       unsigned saved_fvars_size = m_fvars.size();
12561       if (!is_let(e)) e = ensure_terminal(e);
12562       e = visit(e);
12563       expr r = mk_let(saved_fvars_size, e);
12564       lean_assert(!is_lambda(r));
12565       return m_lctx.mk_lambda(binding_fvars, r);
12566   }
12567
12568   expr mk_let_decl(expr const &type, expr const &e) {
12569       expr fvar = m_lctx.mk_local_decl(ngen(), next_name(), type, e);
12570       m_fvars.push_back(fvar);
12571       return fvar;
12572   }
12573
12574   expr mk_sproj(expr const &major, unsigned size, unsigned num,
12575                 unsigned offset) {
12576       return mk_app(mk_llnf_sproj(size, num, offset), major);
12577   }
12578
12579   expr mk_fproj(expr const &major, unsigned num, unsigned offset) {
12580       return mk_app(mk_llnf_fproj(num, offset), major);
12581   }
12582
12583   expr mk_uproj(expr const &major, unsigned idx) {
12584       return mk_app(mk_llnf_uproj(idx), major);
12585   }
12586
12587   expr mk_sset(expr const &major, unsigned size, unsigned num,
12588                unsigned offset, expr const &v) {
12589       return mk_app(mk_llnf_sset(size, num, offset), major, v);
12590   }
12591
12592   expr mk_fset(expr const &major, unsigned num, unsigned offset,
12593                expr const &v) {
12594       return mk_app(mk_llnf_fset(num, offset), major, v);
12595   }
12596
12597   expr mk_uset(expr const &major, unsigned idx, expr const &v) {
12598       return mk_app(mk_llnf_uset(idx), major, v);
12599   }
```

```
12600
12601     expr visit_cases(expr const &e) {
12602         buffer<expr> args;
12603         expr const &fn = get_app_args(e, args);
12604         lean_assert(is_constant(fn));
12605         name const &I_name = const_name(fn).get_prefix();
12606         if (is_inductive_predicate(env(), I_name))
12607             throw exception(sstream()
12608                             << "code generation failed, inductive predicate '"
12609                             << I_name << "' is not supported");
12610         buffer<name> cnames;
12611         get_constructor_names(env(), I_name, cnames);
12612         lean_assert(args.size() == cnames.size() + 1);
12613         /* Process major premise */
12614         expr major = visit(args[0]);
12615         args[0] = major;
12616         expr reachable_case;
12617         unsigned num_reachable = 0;
12618         expr some_reachable;
12619         /* We use `is_id` to track whether this "g_cases_on"-application is of
12620            the form
12621            ```
12622            C.cases_on major (fun ..., _cnstr.0.0) ... (fun ..., _cnstr.(n-1).0)
12623            ```
12624            This kind of application reduces to `major`. This optimization is
12625            useful for code such as:
12626            ```
12627            @decidable.cases_on t _cnstr.0.0 _cnstr.1.0
12628            ```
12629            which reduces to `t`.
12630
12631            TODO(Leo): extend `is_id` when there multiple nested cases_on
12632            applications. Example:
12633            ```
12634            @prod.cases_on _x_1 (λ fst snd,
12635              @except.cases_on fst
12636                (λ a, let _x_2 := except.error ■ ■ a in (_x_2, snd))
12637                (λ a, let _x_3 := except.ok ■ ■ a in (_x_3, snd)))
12638            ```
12639         */
12640         bool is_id = true;
12641         // bool all_eq = true;
12642         /* Process minor premises */
12643         for (unsigned i = 0; i < cnames.size(); i++) {
12644             unsigned saved_fvars_size = m_fvars.size();
12645             expr minor = args[i + 1];
12646             if (minor == mk_enf_neutral()) {
12647                 // This can happen when a branch returns a proposition
12648                 num_reachable++;
12649                 some_reachable = minor;
12650             } else {
12651                 cnstr_info cinfo = get_cnstr_info(cnames[i]);
12652                 buffer<expr> fields;
12653                 for (field_info const &info : cinfo.m_field_info) {
12654                     lean_assert(is_lambda(minor));
12655                     switch (info.m_kind) {
12656                         case field_info::Irrelevant:
12657                             fields.push_back(mk_enf_neutral());
12658                             break;
12659                         case field_info::Object:
12660                             fields.push_back(mk_let_decl(
12661                                 mk_enf_object_type(),
12662                                 mk_app(mk_llnf_proj(info.m_idx), major)));
12663                             break;
12664                         case field_info::USize:
12665                             fields.push_back(mk_let_decl(
12666                                 info.get_type(), mk_uproj(major, info.m_idx)));
12667                             break;
12668                         case field_info::Scalar:
12669                             if (info.is_float()) {
```

```
12670                                 fields.push_back(mk_let_decl(
12671                                     info.get_type(), mk_fproj(major, info.m_idx,
12672                                                               info.m_offset)));
12673                             } else {
12674                                 fields.push_back(mk_let_decl(
12675                                     info.get_type(),
12676                                     mk_sproj(major, info.m_size, info.m_idx,
12677                                              info.m_offset)));
12678                             }
12679                             break;
12680                         }
12681                         minor = binding_body(minor);
12682                     }
12683                     minor = instantiate_rev(minor, fields.size(), fields.data());
12684                     if (!is_let(minor)) minor = ensure_terminal(minor);
12685                     minor = visit(minor);
12686                     if (!is_enf_unreachable(minor)) {
12687                         /* If `minor` is not the constructor `i`, then this
12688                          * "g_cases_on" application is not the identity. */
12689                         unsigned cidx, nusizes, ssz;
12690                         if (!(is_llnf_cnstr(minor, cidx, nusizes, ssz) &&
12691                               cidx == i && nusizes == 0 && ssz == 0)) {
12692                             is_id = false;
12693                         }
12694                         minor = mk_let(saved_fvars_size, minor);
12695 #if 0  // See comment below
12696                         if (num_reachable > 0 && minor != some_reachable) {
12697                             all_eq = false;
12698                         }
12699 #endif
12700                         some_reachable = minor;
12701                         args[i + 1] = minor;
12702                         num_reachable++;
12703                     } else {
12704                         args[i + 1] = minor;
12705                     }
12706                 }
12707             }
12708             if (num_reachable == 0) {
12709                 return mk_enf_unreachable();
12710             } else if (is_id) {
12711                 return major;
12712                 /*
12713                   We remove 1-reachable cases-expressions and all_eq reachable
12714                   later. Reason: `insert_reset_reuse_fn` uses `cases_on`
12715                   applications retrieve constructor layouts.
12716                 */
12717 #if 0
12718             } else if (num_reachable == 1) {
12719                 return some_reachable;
12720             } else if (all_eq) {
12721                 expr r = some_reachable;
12722                 /* Flat `r` if it is a let-declaration */
12723                 buffer<expr> fvars;
12724                 while (is_let(r)) {
12725                     expr val  = instantiate_rev(let_value(r), fvars.size(), fvars.data());
12726                     expr fvar = m_lctx.mk_local_decl(ngen(), let_name(r), let_type(r), val);
12727                     fvars.push_back(fvar);
12728                     m_fvars.push_back(fvar);
12729                     r = let_body(r);
12730                 }
12731                 return instantiate_rev(r, fvars.size(), fvars.data());
12732 #endif
12733             } else {
12734                 return mk_app(fn, args);
12735             }
12736         }
12737
12738         expr visit_constructor(expr const &e) {
12739             buffer<expr> args;
```

```
12740         expr const &k = get_app_args(e, args);
12741         lean_assert(is_constant(k));
12742         if (is_extern_constant(env(), const_name(k)))
12743             return visit_app_default(e);
12744         constructor_val k_val = env().get(const_name(k)).to_constructor_val();
12745         cnstr_info k_info = get_cnstr_info(const_name(k));
12746         unsigned nparams = k_val.get_nparams();
12747         unsigned cidx = k_info.m_cidx;
12748         name const &I = const_name(k).get_prefix();
12749         if (optional<unsigned> r = ::lean::is_enum_type(env(), I)) {
12750             /* We use a literal for enumeration types. */
12751             expr x = mk_let_decl(*to_uint_type(*r), mk_lit(literal(nat(cidx))));
12752             return x;
12753         }
12754         buffer<expr> obj_args;
12755         unsigned j = nparams;
12756         for (field_info const &info : k_info.m_field_info) {
12757             if (info.m_kind != field_info::Irrelevant) args[j] = visit(args[j]);
12758
12759             if (info.m_kind == field_info::Object) {
12760                 obj_args.push_back(args[j]);
12761             }
12762             j++;
12763         }
12764         expr r = mk_app(
12765             mk_llnf_cnstr(I, cidx, k_info.m_num_usizes, k_info.m_scalar_sz),
12766             obj_args);
12767         j = nparams;
12768         bool first = true;
12769         for (field_info const &info : k_info.m_field_info) {
12770             switch (info.m_kind) {
12771                 case field_info::Scalar:
12772                     if (first) {
12773                         r = mk_let_decl(mk_enf_object_type(), r);
12774                     }
12775                     if (info.is_float()) {
12776                         r = mk_let_decl(
12777                             mk_enf_object_type(),
12778                             mk_fset(r, info.m_idx, info.m_offset, args[j]));
12779                     } else {
12780                         r = mk_let_decl(mk_enf_object_type(),
12781                                         mk_sset(r, info.m_size, info.m_idx,
12782                                                 info.m_offset, args[j]));
12783                     }
12784                     first = false;
12785                     break;
12786                 case field_info::USize:
12787                     if (first) {
12788                         r = mk_let_decl(mk_enf_object_type(), r);
12789                     }
12790                     lean_assert(j < args.size());
12791                     r = mk_let_decl(mk_enf_object_type(),
12792                                     mk_uset(r, info.m_idx, args[j]));
12793                     first = false;
12794                     break;
12795
12796                 default:
12797                     break;
12798             }
12799             j++;
12800         }
12801         return r;
12802     }
12803
12804     expr visit_proj(expr const &e) {
12805         name S_name = proj_sname(e);
12806         inductive_val S_val = env().get(S_name).to_inductive_val();
12807         lean_assert(S_val.get_ncnstrs() == 1);
12808         name k_name = head(S_val.get_cnstrs());
12809         cnstr_info k_info = get_cnstr_info(k_name);
```

```
12810         unsigned i = 0;
12811         for (field_info const &info : k_info.m_field_info) {
12812             switch (info.m_kind) {
12813                 case field_info::Irrelevant:
12814                     if (proj_idx(e) == i) return mk_enf_neutral();
12815                     break;
12816                 case field_info::Object:
12817                     if (proj_idx(e) == i)
12818                         return mk_app(mk_llnf_proj(info.m_idx),
12819                                       visit(proj_expr(e)));
12820                     break;
12821                 case field_info::USize:
12822                     if (proj_idx(e) == i)
12823                         return mk_app(mk_llnf_uproj(info.m_idx),
12824                                       visit(proj_expr(e)));
12825                     break;
12826                 case field_info::Scalar:
12827                     if (proj_idx(e) == i) {
12828                         if (info.is_float()) {
12829                             return mk_fproj(visit(proj_expr(e)), info.m_idx,
12830                                             info.m_offset);
12831                         } else {
12832                             return mk_sproj(visit(proj_expr(e)), info.m_size,
12833                                             info.m_idx, info.m_offset);
12834                         }
12835                     }
12836                     break;
12837             }
12838             i++;
12839         }
12840         lean_unreachable();
12841     }
12842
12843     expr visit_constant(expr const &e) {
12844         if (is_constructor(env(), const_name(e))) {
12845             return visit_constructor(e);
12846         } else if (is_enf_neutral(e) || is_enf_unreachable(e) ||
12847                    is_llnf_op(e)) {
12848             return e;
12849         } else {
12850             unsigned arity = get_arity(const_name(e));
12851             if (arity == 0) {
12852                 return e;
12853             } else {
12854                 return mk_app(mk_llnf_closure(), e);
12855             }
12856         }
12857     }
12858
12859     expr visit_app_default(expr const &e) {
12860         buffer<expr> args;
12861         expr const &fn = get_app_args(e, args);
12862         for (expr &arg : args) arg = visit(arg);
12863         return mk_llnf_app(fn, args);
12864     }
12865
12866     expr visit_app(expr const &e) {
12867         expr const &fn = get_app_fn(e);
12868         if (is_cases_on_app(env(), e)) {
12869             return visit_cases(e);
12870         } else if (is_constructor_app(env(), e)) {
12871             return visit_constructor(e);
12872         } else if (is_llnf_op(fn)) {
12873             return e;
12874         } else {
12875             return visit_app_default(e);
12876         }
12877     }
12878
12879     expr visit(expr const &e) {
```

```
12880         switch (e.kind()) {
12881             case expr_kind::App:
12882                 return visit_app(e);
12883             case expr_kind::Lambda:
12884                 return visit_lambda(e);
12885             case expr_kind::Let:
12886                 return visit_let(e);
12887             case expr_kind::Proj:
12888                 return visit_proj(e);
12889             case expr_kind::Const:
12890                 return visit_constant(e);
12891             default:
12892                 return e;
12893         }
12894     }
12895
12896    public:
12897     to_lambda_pure_fn(environment const &env)
12898         : m_st(env), m_x("_x"), m_j("j") {}
12899
12900     expr operator()(expr e) {
12901         if (!is_lambda(e) && !is_let(e)) e = ensure_terminal(e);
12902         expr r = visit(e);
12903         return mk_let(0, r);
12904     }
12905 };
12906
12907 expr get_constant_ll_type(environment const &env, name const &c) {
12908     if (optional<expr> type = get_extern_constant_ll_type(env, c)) {
12909         return *type;
12910     } else {
12911         return env.get(mk_cstage2_name(c)).get_type();
12912     }
12913 }
12914
12915 environment compile_ir(environment const &env, options const &opts,
12916                        comp_decls const &ds) {
12917     buffer<comp_decl> new_ds;
12918     for (comp_decl const &d : ds) {
12919         expr new_v = to_lambda_pure_fn(env)(d.snd());
12920         new_ds.push_back(comp_decl(d.fst(), new_v));
12921     }
12922     return ir::compile(env, opts, new_ds);
12923 }
12924
12925 void initialize_llnf() {
12926     g_apply = new expr(mk_constant("_apply"));
12927     mark_persistent(g_apply->raw());
12928     g_closure = new expr(mk_constant("_closure"));
12929     mark_persistent(g_closure->raw());
12930     g_reuse = new name("_reuse");
12931     mark_persistent(g_reuse->raw());
12932     g_reset = new name("_reset");
12933     mark_persistent(g_reset->raw());
12934     g_sset = new name("_sset");
12935     mark_persistent(g_sset->raw());
12936     g_fset = new name("_fset");
12937     mark_persistent(g_fset->raw());
12938     g_uset = new name("_uset");
12939     mark_persistent(g_uset->raw());
12940     g_proj = new name("_proj");
12941     mark_persistent(g_proj->raw());
12942     g_sproj = new name("_sproj");
12943     mark_persistent(g_sproj->raw());
12944     g_fproj = new name("_sproj");
12945     mark_persistent(g_fproj->raw());
12946     g_uproj = new name("_uproj");
12947     mark_persistent(g_uproj->raw());
12948     g_jmp = new expr(mk_constant("_jmp"));
12949     mark_persistent(g_jmp->raw());
```

```
12950     g_box = new name("_box");
12951     mark_persistent(g_box->raw());
12952     g_unbox = new name("_unbox");
12953     mark_persistent(g_unbox->raw());
12954     g_inc = new expr(mk_constant("_inc"));
12955     mark_persistent(g_inc->raw());
12956     g_dec = new expr(mk_constant("_dec"));
12957     mark_persistent(g_dec->raw());
12958     register_trace_class({"compiler", "lambda_pure"});
12959 }
12960
12961 void finalize_llnf() {
12962     delete g_closure;
12963     delete g_apply;
12964     delete g_reuse;
12965     delete g_reset;
12966     delete g_sset;
12967     delete g_fset;
12968     delete g_proj;
12969     delete g_sproj;
12970     delete g_fproj;
12971     delete g_uset;
12972     delete g_uproj;
12973     delete g_jmp;
12974     delete g_box;
12975     delete g_unbox;
12976     delete g_inc;
12977     delete g_dec;
12978 }
12979 }  // namespace lean
12980 // ::::::::::::::
12981 // compiler/reduce_arity.cpp
12982 // ::::::::::::::
12983 /*
12984 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
12985 Released under Apache 2.0 license as described in the file LICENSE.
12986 Author: Leonardo de Moura
12987 */
12988 #include "kernel/instantiate.h"
12989 #include "library/compiler/export_attribute.h"
12990 #include "library/compiler/util.h"
12991 #include "library/util.h"
12992
12993 namespace lean {
12994 #define REDUCE_ARITY_SUFFIX "_rarg"
12995
12996 name mk_reduce_arity_aux_fn(name const &n) {
12997     return name(n, REDUCE_ARITY_SUFFIX);
12998 }
12999
13000 bool is_reduce_arity_aux_fn(name const &n) {
13001     return n.is_string() && !n.is_atomic() &&
13002            strcmp(n.get_string().data(), REDUCE_ARITY_SUFFIX) == 0;
13003 }
13004
13005 bool arity_was_reduced(comp_decl const &cdecl) {
13006     expr v = cdecl.snd();
13007     while (is_lambda(v)) v = binding_body(v);
13008     expr const &f = get_app_fn(v);
13009     if (!is_constant(f)) return false;
13010     name const &n = const_name(f);
13011     return is_reduce_arity_aux_fn(n) && n.get_prefix() == cdecl.fst();
13012 }
13013
13014 comp_decls reduce_arity(environment const &env, comp_decl const &cdecl) {
13015     if (has_export_name(env, cdecl.fst()) || cdecl.fst() == "main") {
13016         /* We do not modify the arity of entry points (i.e., functions with
13017          * attribute [export]) */
13018         return comp_decls(cdecl);
13019     }
```

```
13020     expr code = cdecl.snd();
13021     buffer<expr> fvars;
13022     name_generator ngen;
13023     local_ctx lctx;
13024     while (is_lambda(code)) {
13025         lean_assert(!has_loose_bvars(binding_domain(code)));
13026         expr fvar =
13027             lctx.mk_local_decl(ngen, binding_name(code), binding_domain(code));
13028         fvars.push_back(fvar);
13029         code = binding_body(code);
13030     }
13031     code = instantiate_rev(code, fvars.size(), fvars.data());
13032     buffer<expr> new_fvars;
13033 #if 1
13034     /* For now, we remove just the prefix.
13035        Removing unused variables that occur in other parts of the declaration
13036        seem to create problems. Example: we may create more closures if the
13037        function is partially applied. By eliminating just a prefix, we get the
13038        most common case: a function that starts with a sequence of type
13039        variables.
13040        TODO(Leo): improve this. */
13041     bool found_used = false;
13042     for (expr &fvar : fvars) {
13043         if (found_used || has_fvar(code, fvar)) {
13044             found_used = true;
13045             new_fvars.push_back(fvar);
13046         }
13047     }
13048 #else
13049     for (expr &fvar : fvars) {
13050         if (has_fvar(code, fvar)) {
13051             new_fvars.push_back(fvar);
13052         }
13053     }
13054 #endif
13055     if (fvars.size() == new_fvars.size() || new_fvars.empty()) {
13056         /* Do nothing if:
13057            1- All arguments are used.
13058            2- No argument was used, and auxiliary declaration would be a
13059            constant. This is not safe since constants are executed during
13060            initialization, and we may execute unreachable code when one of the
13061            "unused" arguments is an uninhabited type. Here is an example where
13062            the auxiliary definition would be a constant:
13063
13064               ```
13065               def false.elim {C : Sort u} (h : false) : C := ..
13066               ```
13067         */
13068         return comp_decls(cdecl);
13069     }
13070     name red_fn = mk_reduce_arity_aux_fn(cdecl.fst());
13071     expr red_code = lctx.mk_lambda(new_fvars, code);
13072     comp_decl red_decl(red_fn, red_code);
13073     /* Replace `cdecl` code with a call to `red_fn`.
13074        We rely on inlining to reduce calls to `cdecl` into calls to `red_decl`.
13075      */
13076     expr new_code = mk_app(mk_constant(red_fn), new_fvars);
13077     new_code = try_eta(lctx.mk_lambda(fvars, new_code));
13078     comp_decl new_decl(cdecl.fst(), new_code);
13079     return comp_decls(red_decl, comp_decls(new_decl));
13080 }
13081
13082 comp_decls reduce_arity(environment const &env, comp_decls const &ds) {
13083     comp_decls r;
13084     for (comp_decl const &d : ds) {
13085         r = append(r, reduce_arity(env, d));
13086     }
13087     return r;
13088 }
13089 }  // namespace lean
```

```
// ::::::::::::::
// :compiler/simp_app_args.cpp
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include "kernel/instantiate.h"
#include "library/compiler/ll_infer_type.h"
#include "library/compiler/util.h"

namespace lean {
/* Make sure every argument of applications and projections is a free variable
 * (or neutral element). */
class simp_app_args_fn {
    type_checker::state m_st;
    local_ctx m_lctx;
    buffer<expr> m_fvars;
    name m_x;
    unsigned m_next_idx{1};

    environment const &env() const { return m_st.env(); }
    name_generator &ngen() { return m_st.ngen(); }

    name next_name() {
        name r = m_x.append_after(m_next_idx);
        m_next_idx++;
        return r;
    }

    expr mk_let(unsigned saved_fvars_size, expr r) {
        lean_assert(saved_fvars_size <= m_fvars.size());
        if (saved_fvars_size == m_fvars.size()) return r;
        r = m_lctx.mk_lambda(m_fvars.size() - saved_fvars_size,
                             m_fvars.data() + saved_fvars_size, r);
        m_fvars.shrink(saved_fvars_size);
        return r;
    }

    expr visit_let(expr e) {
        buffer<expr> curr_fvars;
        while (is_let(e)) {
            lean_assert(!has_loose_bvars(let_type(e)));
            expr t = let_type(e);
            expr v = visit(instantiate_rev(let_value(e), curr_fvars.size(),
                                           curr_fvars.data()));
            name n = let_name(e);
            /* Pseudo "do" joinpoints are used to implement a temporary HACK.
             * See `visit_let` method at `lcnf.cpp` */
            if (is_internal_name(n) && !is_join_point_name(n) &&
                !is_pseudo_do_join_point_name(n)) {
                n = next_name();
            }
            expr fvar = m_lctx.mk_local_decl(ngen(), n, t, v);
            curr_fvars.push_back(fvar);
            m_fvars.push_back(fvar);
            e = let_body(e);
        }
        return visit(instantiate_rev(e, curr_fvars.size(), curr_fvars.data()));
    }

    expr visit_lambda(expr e) {
        buffer<expr> binding_fvars;
        while (is_lambda(e)) {
            lean_assert(!has_loose_bvars(binding_domain(e)));
            expr new_fvar = m_lctx.mk_local_decl(
                ngen(), binding_name(e), binding_domain(e), binding_info(e));
            binding_fvars.push_back(new_fvar);
```

```
13160             e = binding_body(e);
13161         }
13162         e = instantiate_rev(e, binding_fvars.size(), binding_fvars.data());
13163         unsigned saved_fvars_size = m_fvars.size();
13164         expr r = mk_let(saved_fvars_size, visit(e));
13165         lean_assert(!is_lambda(r));
13166         return m_lctx.mk_lambda(binding_fvars, r);
13167     }
13168
13169     expr ensure_simple_arg(expr const &e) {
13170         if (is_fvar(e) || is_enf_neutral(e)) {
13171             return e;
13172         } else if (is_lit(e)) {
13173             expr fvar = m_lctx.mk_local_decl(ngen(), next_name(),
13174                                              mk_enf_object_type(), e);
13175             m_fvars.push_back(fvar);
13176             return fvar;
13177         } else if (is_constant(e)) {
13178             expr type = ll_infer_type(env(), e);
13179             expr fvar = m_lctx.mk_local_decl(ngen(), next_name(), type, e);
13180             m_fvars.push_back(fvar);
13181             return fvar;
13182         } else {
13183             lean_unreachable();
13184         }
13185     }
13186
13187     expr visit_proj(expr const &e) {
13188         expr arg = ensure_simple_arg(proj_expr(e));
13189         return update_proj(e, arg);
13190     }
13191
13192     expr visit_app(expr const &e) {
13193         buffer<expr> args;
13194         expr const &fn = get_app_args(e, args);
13195         if (is_cases_on_app(env(), e)) {
13196             args[0] = ensure_simple_arg(args[0]);
13197             for (unsigned i = 1; i < args.size(); i++) {
13198                 if (is_lambda(args[i])) {
13199                     args[i] = visit(args[i]);
13200                 } else {
13201                     unsigned saved_fvars_size = m_fvars.size();
13202                     args[i] = mk_let(saved_fvars_size, visit(args[i]));
13203                 }
13204             }
13205         } else if (is_morally_num_lit(e)) {
13206             /* Do not convert `x := uint*.of_nat <val>` into `y := <val>, x :=
13207              * uint*.of_nat y` */
13208             return e;
13209         } else {
13210             for (expr &arg : args) arg = ensure_simple_arg(arg);
13211         }
13212         return mk_app(fn, args);
13213     }
13214
13215     expr visit(expr const &e) {
13216         switch (e.kind()) {
13217             case expr_kind::App:
13218                 return visit_app(e);
13219             case expr_kind::Lambda:
13220                 return visit_lambda(e);
13221             case expr_kind::Let:
13222                 return visit_let(e);
13223             case expr_kind::Proj:
13224                 return visit_proj(e);
13225             default:
13226                 return e;
13227         }
13228     }
13229
```

```
13230    public:
13231     simp_app_args_fn(environment const &env) : m_st(env), m_x("_x") {}
13232
13233     expr operator()(expr const &e) { return mk_let(0, visit(e)); }
13234 };
13235
13236 expr simp_app_args(environment const &env, expr const &e) {
13237     return simp_app_args_fn(env)(e);
13238 }
13239 }  // namespace lean
13240 // ::::::::::::::
13241 // :compiler/specialize.cpp
13242 // ::::::::::::::
13243 /*
13244 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
13245 Released under Apache 2.0 license as described in the file LICENSE.
13246
13247 Author: Leonardo de Moura
13248 */
13249 #include <lean/flet.h>
13250
13251 #include <algorithm>
13252
13253 #include "kernel/abstract.h"
13254 #include "kernel/for_each_fn.h"
13255 #include "kernel/instantiate.h"
13256 #include "library/class.h"
13257 #include "library/compiler/csimp.h"
13258 #include "library/compiler/util.h"
13259 #include "library/trace.h"
13260
13261 namespace lean {
13262 extern "C" uint8 lean_has_specialize_attribute(object *env, object *n);
13263 extern "C" uint8 lean_has_nospecialize_attribute(object *env, object *n);
13264
13265 bool has_specialize_attribute(environment const &env, name const &n) {
13266     return lean_has_specialize_attribute(env.to_obj_arg(), n.to_obj_arg());
13267 }
13268
13269 bool has_nospecialize_attribute(environment const &env, name const &n) {
13270     return lean_has_nospecialize_attribute(env.to_obj_arg(), n.to_obj_arg());
13271 }
13272
13273 /* IMPORTANT: We currently do NOT specialize Fixed arguments.
13274    Only FixedNeutral, FixedHO and FixedInst.
13275    We do not have good heuristics to decide when it is a good idea to do it.
13276    TODO(Leo): allow users to specify that they want to consider some Fixed
13277    arguments for specialization.
13278 */
13279 enum class spec_arg_kind {
13280     Fixed,
13281     FixedNeutral, /* computationally neutral */
13282     FixedHO,      /* higher order */
13283     FixedInst,    /* type class instance */
13284     Other
13285 };
13286
13287 static spec_arg_kind to_spec_arg_kind(object_ref const &r) {
13288     lean_assert(is_scalar(r.raw()));
13289     return static_cast<spec_arg_kind>(unbox(r.raw()));
13290 }
13291 typedef objects spec_arg_kinds;
13292 static spec_arg_kinds to_spec_arg_kinds(buffer<spec_arg_kind> const &ks) {
13293     spec_arg_kinds r;
13294     unsigned i = ks.size();
13295     while (i > 0) {
13296         --i;
13297         r = spec_arg_kinds(object_ref(box(static_cast<unsigned>(ks[i]))), r);
13298     }
13299     return r;
```

```
13300 }
13301 static void to_buffer(spec_arg_kinds const &ks, buffer<spec_arg_kind> &r) {
13302     for (object_ref const &k : ks) {
13303         r.push_back(to_spec_arg_kind(k));
13304     }
13305 }
13306
13307 static bool has_fixed_inst_arg(buffer<spec_arg_kind> const &ks) {
13308     for (spec_arg_kind k : ks) {
13309         if (k == spec_arg_kind::FixedInst) return true;
13310     }
13311     return false;
13312 }
13313
13314 /* Return true if `ks` contains kind != Other */
13315 static bool has_kind_ne_other(buffer<spec_arg_kind> const &ks) {
13316     for (spec_arg_kind k : ks) {
13317         if (k != spec_arg_kind::Other) return true;
13318     }
13319     return false;
13320 }
13321
13322 char const *to_str(spec_arg_kind k) {
13323     switch (k) {
13324         case spec_arg_kind::Fixed:
13325             return "F";
13326         case spec_arg_kind::FixedNeutral:
13327             return "N";
13328         case spec_arg_kind::FixedHO:
13329             return "H";
13330         case spec_arg_kind::FixedInst:
13331             return "I";
13332         case spec_arg_kind::Other:
13333             return "X";
13334     }
13335     lean_unreachable();
13336 }
13337
13338 class spec_info : public object_ref {
13339    public:
13340     spec_info(names const &ns, spec_arg_kinds ks)
13341         : object_ref(mk_cnstr(0, ns, ks)) {}
13342     spec_info() : spec_info(names(), spec_arg_kinds()) {}
13343     spec_info(spec_info const &other) : object_ref(other) {}
13344     spec_info(spec_info &&other) : object_ref(other) {}
13345     spec_info(b_obj_arg o, bool b) : object_ref(o, b) {}
13346     spec_info &operator=(spec_info const &other) {
13347         object_ref::operator=(other);
13348         return *this;
13349     }
13350     spec_info &operator=(spec_info &&other) {
13351         object_ref::operator=(other);
13352         return *this;
13353     }
13354     names const &get_mutual_decls() const {
13355         return static_cast<names const &>(cnstr_get_ref(*this, 0));
13356     }
13357     spec_arg_kinds const &get_arg_kinds() const {
13358         return static_cast<spec_arg_kinds const &>(cnstr_get_ref(*this, 1));
13359     }
13360     void serialize(serializer &s) const { s.write_object(raw()); }
13361     static spec_info deserialize(deserializer &d) {
13362         return spec_info(d.read_object(), true);
13363     }
13364 };
13365
13366 extern "C" object *lean_add_specialization_info(object *env, object *fn,
13367                                                 object *info);
13368 extern "C" object *lean_get_specialization_info(object *env, object *fn);
13369
```

```
13370 static environment save_specialization_info(environment const &env,
13371                                               name const &fn,
13372                                               spec_info const &si) {
13373     return environment(lean_add_specialization_info(
13374         env.to_obj_arg(), fn.to_obj_arg(), si.to_obj_arg()));
13375 }
13376
13377 static optional<spec_info> get_specialization_info(environment const &env,
13378                                                    name const &fn) {
13379     return to_optional<spec_info>(
13380         lean_get_specialization_info(env.to_obj_arg(), fn.to_obj_arg()));
13381 }
13382
13383 typedef buffer<pair<name, buffer<spec_arg_kind>>> spec_info_buffer;
13384
13385 /* We only specialize arguments that are "fixed" in mutual recursive
13386    declarations. The buffer `info_buffer` stores which arguments are fixed for
13387    each declaration in a mutual recursive declaration. This procedure traverses
13388    `e` and updates `info_buffer`.
13389
13390    Remark: we only create free variables for the header of each declaration.
13391    Then, we assume an argument of a recursive call is fixed iff it is a free
13392    variable (see `update_spec_info`). */
13393 static void update_info_buffer(environment const &env, expr e,
13394                                name_set const &S,
13395                                spec_info_buffer &info_buffer) {
13396     while (true) {
13397         switch (e.kind()) {
13398             case expr_kind::Lambda:
13399                 e = binding_body(e);
13400                 break;
13401             case expr_kind::Let:
13402                 update_info_buffer(env, let_value(e), S, info_buffer);
13403                 e = let_body(e);
13404                 break;
13405             case expr_kind::App:
13406                 if (is_cases_on_app(env, e)) {
13407                     buffer<expr> args;
13408                     expr const &c_fn = get_app_args(e, args);
13409                     unsigned minors_begin;
13410                     unsigned minors_end;
13411                     std::tie(minors_begin, minors_end) =
13412                         get_cases_on_minors_range(env, const_name(c_fn));
13413                     for (unsigned i = minors_begin; i < minors_end; i++) {
13414                         update_info_buffer(env, args[i], S, info_buffer);
13415                     }
13416                 } else {
13417                     buffer<expr> args;
13418                     expr const &fn = get_app_args(e, args);
13419                     if (is_constant(fn) && S.contains(const_name(fn))) {
13420                         for (auto &entry : info_buffer) {
13421                             if (entry.first == const_name(fn)) {
13422                                 unsigned sz = entry.second.size();
13423                                 for (unsigned i = 0; i < sz; i++) {
13424                                     if (i >= args.size() || !is_fvar(args[i])) {
13425                                         entry.second[i] = spec_arg_kind::Other;
13426                                     }
13427                                 }
13428                                 break;
13429                             }
13430                         }
13431                     }
13432                 }
13433                 return;
13434             default:
13435                 return;
13436         }
13437     }
13438 }
13439
```

```
environment update_spec_info(environment const &env, comp_decls const &ds) {
    name_set S;
    spec_info_buffer d_infos;
    name_generator ngen;
    /* Initialzie d_infos and S */
    for (comp_decl const &d : ds) {
        S.insert(d.fst());
        d_infos.push_back(pair<name, buffer<spec_arg_kind>>());
        auto &info = d_infos.back();
        info.first = d.fst();
        expr code = d.snd();
        buffer<expr> fvars;
        local_ctx lctx;
        while (is_lambda(code)) {
            expr type = instantiate_rev(binding_domain(code), fvars.size(),
                                        fvars.data());
            expr fvar = lctx.mk_local_decl(ngen, binding_name(code), type);
            fvars.push_back(fvar);
            if (is_inst_implicit(binding_info(code))) {
                info.second.push_back(spec_arg_kind::FixedInst);
            } else {
                type_checker tc(env, lctx);
                type = tc.whnf(type);
                if (is_sort(type) || tc.is_prop(type)) {
                    info.second.push_back(spec_arg_kind::FixedNeutral);
                } else if (is_pi(type)) {
                    while (is_pi(type)) {
                        expr fvar = lctx.mk_local_decl(ngen, binding_name(type),
                                                       binding_domain(type));
                        type = type_checker(env, lctx).whnf(
                            instantiate(binding_body(type), fvar));
                    }
                    if (is_sort(type)) {
                        /* Functions that return types are not relevant */
                        info.second.push_back(spec_arg_kind::FixedNeutral);
                    } else {
                        info.second.push_back(spec_arg_kind::FixedHO);
                    }
                } else {
                    info.second.push_back(spec_arg_kind::Fixed);
                }
            }
            code = binding_body(code);
        }
    }
    /* Update d_infos */
    name x("_x");
    for (comp_decl const &d : ds) {
        buffer<expr> fvars;
        expr code = d.snd();
        unsigned i = 1;
        /* Create free variables for header variables. */
        while (is_lambda(code)) {
            fvars.push_back(mk_fvar(name(x, i)));
            code = binding_body(code);
        }
        code = instantiate_rev(code, fvars.size(), fvars.data());
        update_info_buffer(env, code, S, d_infos);
    }
    /* Update extension */
    environment new_env = env;
    names mutual_decls =
        map2<name>(ds, [&](comp_decl const &d) { return d.fst(); });
    for (pair<name, buffer<spec_arg_kind>> const &info : d_infos) {
        name const &n = info.first;
        spec_info si(mutual_decls, to_spec_arg_kinds(info.second));
        lean_trace(
            name({"compiler", "spec_info"}), tout() << n; for (spec_arg_kind k
                                                               : info.second) {
                tout() << " " << to_str(k);
```

```
            } tout() << "\n";);
        new_env = save_specialization_info(new_env, n, si);
    }
    return new_env;
}

extern "C" object *lean_cache_specialization(object *env, object *e,
                                             object *fn);
extern "C" object *lean_get_cached_specialization(object *env, object *e);

static environment cache_specialization(environment const &env, expr const &k,
                                        name const &fn) {
    return environment(lean_cache_specialization(
        env.to_obj_arg(), k.to_obj_arg(), fn.to_obj_arg()));
}

static optional<name> get_cached_specialization(environment const &env,
                                                expr const &e) {
    return to_optional<name>(
        lean_get_cached_specialization(env.to_obj_arg(), e.to_obj_arg()));
}

class specialize_fn {
    type_checker::state m_st;
    csimp_cfg m_cfg;
    local_ctx m_lctx;
    buffer<comp_decl> m_new_decls;
    name m_base_name;
    name m_at;
    name m_spec;
    unsigned m_next_idx{1};
    name_set m_to_respecialize;

    environment const &env() { return m_st.env(); }

    name_generator &ngen() { return m_st.ngen(); }

    expr visit_lambda(expr e) {
        flet<local_ctx> save_lctx(m_lctx, m_lctx);
        buffer<expr> fvars;
        while (is_lambda(e)) {
            expr new_type =
                instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
            expr new_fvar =
                m_lctx.mk_local_decl(ngen(), binding_name(e), new_type);
            fvars.push_back(new_fvar);
            e = binding_body(e);
        }
        expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
        return m_lctx.mk_lambda(fvars, r);
    }

    expr visit_let(expr e) {
        flet<local_ctx> save_lctx(m_lctx, m_lctx);
        buffer<expr> fvars;
        while (is_let(e)) {
            expr new_type =
                instantiate_rev(let_type(e), fvars.size(), fvars.data());
            expr new_val = visit(
                instantiate_rev(let_value(e), fvars.size(), fvars.data()));
            expr new_fvar =
                m_lctx.mk_local_decl(ngen(), let_name(e), new_type, new_val);
            fvars.push_back(new_fvar);
            e = let_body(e);
        }
        expr r = visit(instantiate_rev(e, fvars.size(), fvars.data()));
        /*
          We eagerly remove dead let-declarations to avoid unnecessary
          dependencies when specializing code. For example, consider the
          following piece of code.
```

```
        ```
        fun (ys : List Nat) (w : IO.RealWorld) =>
        let x_1 : Monad (EIO IO.Error) := ...;
        let x_2 : Monad (StateT Nat IO) := ... x_1 ..;
        let x_3 : Nat → StateT Nat IO Unit := fun (y a : Nat) (w :
        IO.RealWorld) => let x_4 : MonadLift IO (StateT Nat IO) := ... x_1
        ...; let x_5 : MonadIO (StateT Nat IO) := ... x_4 ...; IO.println _
        x_2 x_5 Nat Nat.HasToString y a w; let x_6 : EStateM.Result IO.Error
        IO.RealWorld (Unit × Nat) := List.forM _ x_2 Nat x_3 ys 0 w;
        ...

        ```
        After we specialize `IO.println ...`, we obtain `IO.println.spec y a
        w`. That is, the dependencies have been eliminated. So, by eagerly
        removing the dead let-declarations, we eliminate `x_4` and `x_5`, and
        `x_3` becomes
        ```
        let x_3 : Nat → StateT Nat IO Unit := fun (y a : Nat) (w :
        IO.RealWorld) => IO.println.spec y a w;
        ```
        Now, suppose we haven't eliminated the dependencies. Then, when we try
        to specialize `List.forM _ x_2 Nat x_3 ys 0 w` we will incorrectly
        assume that the binder in `x_3` depends on the let-declaration `x_1`.
        The heuristic for avoiding work duplication (see comment at
        `spec_ctx`) will force the specializer to abstract `x_1`, and `forM`
        will be specialized for an arbitrary `x_1 : Monad (EIO IO.Error)`.

        Another possible solution for this issue is to always copy instances
        at `dep_collector`. However, we may be duplicating work. Note that, we
        don't have here a way to distinguish between let-decls that come from
        inst-implicit arguments from the ones have been manually written by
        users.

        Here is the code that was used to produce the fragment above.
        ```
        def g (ys : List Nat) : IO Nat := do
        let x := 0;
        (_, x) ← StateT.run (ys.forM fun y => IO.println y) x;
        pure x
        ```
        If we don't eagerly remove dead let-declarations, then we can the
        nonoptimal code for the `forM` specialization using `set_option
        trace.compiler.ir.result true`
    */
    return m_lctx.mk_lambda(fvars, r,
                            true /* remove dead let-declarations */);
}

expr visit_cases_on(expr const &e) {
    lean_assert(is_cases_on_app(env(), e));
    buffer<expr> args;
    expr const &c = get_app_args(e, args);
    /* visit minor premises */
    unsigned minor_idx;
    unsigned minors_end;
    std::tie(minor_idx, minors_end) =
        get_cases_on_minors_range(env(), const_name(c));
    for (; minor_idx < minors_end; minor_idx++) {
        args[minor_idx] = visit(args[minor_idx]);
    }
    return mk_app(c, args);
}

expr find(expr const &e) {
    if (is_fvar(e)) {
        if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
            if (optional<expr> v = decl->get_value()) {
                return find(*v);
            }
        }
```

```
        } else if (is_mdata(e)) {
            return find(mdata_expr(e));
        }
        return e;
    }

    struct spec_ctx {
        typedef rb_expr_map<name> cache;
        names m_mutual;
        /* `m_params` contains all variables that must be lambda abstracted in
           the specialization. It may contain let-variables that occurs inside
           of binders. Reason: avoid work duplication.

           Example: suppose we are trying to specialize the following
           map-application.
           ```
           def f2 (n : nat) (xs : list nat) : list (list nat) :=
           let ys := list.repeat 0 n in
           xs.map (λ x, x :: ys)
           ```
           We don't want to copy `list.repeat 0 n` inside of the specialized
           code.

           However, there is one exception: join-points.
           For join-points, there is no risk of work duplication, but we
           tolerate code duplication.
        */
        buffer<expr> m_params;
        /* `m_vars` contains `m_params` plus all let-declarations.

           Remark: we used to keep m_params and let-declarations in separate
           buffers. This produced incorrect results when the type of a variable
           in `m_params` depended on a let-declaration. */
        buffer<expr> m_vars;
        cache m_cache;
        buffer<comp_decl> m_pre_decls;

        bool in_mutual_decl(name const &n) const {
            return std::find(m_mutual.begin(), m_mutual.end(), n) !=
                   m_mutual.end();
        }
    };

    void get_arg_kinds(name const &fn, buffer<spec_arg_kind> &kinds) {
        optional<spec_info> info = get_specialization_info(env(), fn);
        lean_assert(info);
        to_buffer(info->get_arg_kinds(), kinds);
    }

    static void to_bool_mask(buffer<spec_arg_kind> const &kinds, bool has_attr,
                             buffer<bool> &mask) {
        unsigned sz = kinds.size();
        mask.resize(sz, false);
        unsigned i = sz;
        bool found_inst = false;
        bool first = true;
        while (i > 0) {
            --i;
            switch (kinds[i]) {
                case spec_arg_kind::Other:
                    break;
                case spec_arg_kind::FixedInst:
                    mask[i] = true;
                    if (first) mask.shrink(i + 1);
                    first = false;
                    found_inst = true;
                    break;
                case spec_arg_kind::Fixed:
                    // REMARK: We have disabled specialization for this kind of
                    // argument.
```

```
13720                     break;
13721                 case spec_arg_kind::FixedHO:
13722                 case spec_arg_kind::FixedNeutral:
13723                     if (has_attr || found_inst) {
13724                         mask[i] = true;
13725                         if (first) mask.shrink(i + 1);
13726                         first = false;
13727                     }
13728                     break;
13729             }
13730         }
13731     }
13732
13733     bool has_specialize_attribute(name const &fn) {
13734         return ::lean::has_specialize_attribute(env(), fn) ||
13735                m_to_respecialize.contains(fn);
13736     }
13737
13738     void get_bool_mask(name const &fn, unsigned args_size, buffer<bool> &mask) {
13739         buffer<spec_arg_kind> kinds;
13740         get_arg_kinds(fn, kinds);
13741         if (kinds.size() > args_size) kinds.shrink(args_size);
13742         to_bool_mask(kinds, has_specialize_attribute(fn), mask);
13743     }
13744
13745     name mk_spec_name(name const &fn) {
13746         name r = fn + m_at + m_base_name + (m_spec.append_after(m_next_idx));
13747         m_next_idx++;
13748         return r;
13749     }
13750
13751     static expr mk_cache_key(expr const &fn,
13752                              buffer<optional<expr>> const &mask) {
13753         expr r = fn;
13754         for (optional<expr> const &b : mask) {
13755             if (b)
13756                 r = mk_app(r, *b);
13757             else
13758                 r = mk_app(r, expr());
13759         }
13760         return r;
13761     }
13762
13763     bool is_specialize_candidate(expr const &fn, buffer<expr> const &args) {
13764         lean_assert(is_constant(fn));
13765         buffer<spec_arg_kind> kinds;
13766         get_arg_kinds(const_name(fn), kinds);
13767         if (!has_specialize_attribute(const_name(fn)) &&
13768             !has_fixed_inst_arg(kinds))
13769             return false;                               /* Nothing to specialize */
13770         if (!has_kind_ne_other(kinds)) return false; /* Nothing to specialize */
13771         type_checker tc(m_st, m_lctx);
13772         for (unsigned i = 0; i < args.size(); i++) {
13773             if (i >= kinds.size()) break;
13774             spec_arg_kind k = kinds[i];
13775             expr w;
13776             switch (k) {
13777                 case spec_arg_kind::FixedNeutral:
13778                     break;
13779                 case spec_arg_kind::FixedInst:
13780                     /* We specialize this kind of argument if it reduces to a
13781                        constructor application or lambda. Type class instances
13782                        arguments are usually free variables bound to lambda
13783                        declarations, or quickly reduce to constructor
13784                        application or lambda. So, the following `whnf` is
13785                        probably harmless. We need to consider the lambda case
13786                        because of arguments such as `[decidable_rel lt]` */
13787                     w = tc.whnf(args[i]);
13788                     if (is_constructor_app(env(), w) || is_lambda(w))
13789                         return true;
```

```
13790                     break;
13791                 case spec_arg_kind::FixedHO:
13792                     /* We specialize higher-order arguments if they are lambda
13793                        applications or a constant application.
13794
13795                        Remark: it is not feasible to invoke whnf since it may
13796                        consume a lot of time. */
13797                     w = find(args[i]);
13798                     if (is_lambda(w) || is_constant(get_app_fn(w))) return true;
13799                     break;
13800                 case spec_arg_kind::Fixed:
13801                     /* We specialize this kind of argument if they are
13802                        constructor applications or literals. Remark: it is not
13803                        feasible to invoke whnf since it may consume a lot of
13804                        time. */
13805                     break;  // We have disabled this kind of argument
13806                     w = find(args[i]);
13807                     if (is_constructor_app(env(), w) || is_lit(w)) return true;
13808                     break;
13809                 case spec_arg_kind::Other:
13810                     break;
13811             }
13812         }
13813         return false;
13814     }
13815
13816     /* Auxiliary class for collecting specialization dependencies. */
13817     class dep_collector {
13818         local_ctx m_lctx;
13819         name_set m_visited_not_in_binder;
13820         name_set m_visited_in_binder;
13821         spec_ctx &m_ctx;
13822
13823         void collect_fvar(expr const &x, bool in_binder) {
13824             name const &x_name = fvar_name(x);
13825             if (!in_binder) {
13826                 if (m_visited_not_in_binder.contains(x_name)) return;
13827                 m_visited_not_in_binder.insert(x_name);
13828                 local_decl decl = m_lctx.get_local_decl(x);
13829                 optional<expr> v = decl.get_value();
13830                 if (m_visited_in_binder.contains(x_name)) {
13831                     /* If `x` was already visited in context inside of a binder,
13832                        then it is already in `m_ctx.m_vars` and
13833                        `m_ctx.m_params`. */
13834                 } else {
13835                     /* Recall that `m_ctx.m_vars` contains all variables (lambda
13836                        and let) the specialization depends on, and
13837                        `m_ctx.m_params` contains the ones that should be lambda
13838                        abstracted. */
13839                     m_ctx.m_vars.push_back(x);
13840                     /* Thus, a variable occuring outside of a binder is only
13841                        lambda abstracted if it is not a let-variable. */
13842                     if (!v) m_ctx.m_params.push_back(x);
13843                 }
13844                 collect(decl.get_type(), false);
13845                 if (v) collect(*v, false);
13846             } else {
13847                 if (m_visited_in_binder.contains(x_name)) return;
13848                 m_visited_in_binder.insert(x_name);
13849                 local_decl decl = m_lctx.get_local_decl(x);
13850                 optional<expr> v = decl.get_value();
13851                 /* Remark: we must not lambda abstract join points.
13852                    There is no risk of work duplication in this case, only code
13853                    duplication. */
13854                 bool is_jp = is_join_point_name(decl.get_user_name());
13855                 // lean_assert(!v || !is_irrelevant_type(m_st, m_lctx,
13856                 // decl.get_type()));
13857                 if (m_visited_not_in_binder.contains(x_name)) {
13858                     /* If `x` was already visited in a context outside of
13859                        a binder, then it is already in `m_ctx.m_vars`.
```

```
                    If `x` is not a let-variable, then it is also already in
                    `m_ctx.m_params`. */
                if (v && !is_jp) {
                    m_ctx.m_params.push_back(x);
                    v = none_expr(); /* make sure we don't collect v's
                                        dependencies */
                }
            } else {
                /* Recall that if `x` occurs inside of a binder, then it
                   will always be lambda abstracted. Reason: avoid work
                   duplication. Example: suppose we are trying to specialize
                   the following map-application.
                   ```
                   def f2 (n : nat) (xs : list nat) : list (list nat) :=
                   let ys := list.repeat 0 n in
                   xs.map (λ x, x :: ys)
                   ```
                   We don't want to copy `list.repeat 0 n` inside of the
                   specialized code.

                   See comment above about join points.

                   Remark: if `x` is not a let-var, then we must insert it
                   into m_ctx.m_params.
                */
                m_ctx.m_vars.push_back(x);
                if (!v || (v && !is_jp)) {
                    m_ctx.m_params.push_back(x);
                    v = none_expr(); /* make sure we don't collect v's
                                        dependencies */
                }
            }
            collect(decl.get_type(), true);
            if (v) collect(*v, true);
        }
    }

    void collect(expr e, bool in_binder) {
        while (true) {
            if (!has_fvar(e)) return;
            switch (e.kind()) {
                case expr_kind::Lit:
                case expr_kind::BVar:
                case expr_kind::Sort:
                case expr_kind::Const:
                    return;
                case expr_kind::MVar:
                    lean_unreachable();
                case expr_kind::FVar:
                    collect_fvar(e, in_binder);
                    return;
                case expr_kind::App:
                    collect(app_arg(e), in_binder);
                    e = app_fn(e);
                    break;
                case expr_kind::Lambda:
                case expr_kind::Pi:
                    collect(binding_domain(e), in_binder);
                    if (!in_binder) {
                        collect(binding_body(e), true);
                        return;
                    } else {
                        e = binding_body(e);
                        break;
                    }
                case expr_kind::Let:
                    collect(let_type(e), in_binder);
                    collect(let_value(e), in_binder);
                    e = let_body(e);
                    break;
```

```
13930                     case expr_kind::MData:
13931                         e = mdata_expr(e);
13932                         break;
13933                     case expr_kind::Proj:
13934                         e = proj_expr(e);
13935                         break;
13936                 }
13937             }
13938         }
13939
13940        public:
13941         dep_collector(local_ctx const &lctx, spec_ctx &ctx)
13942             : m_lctx(lctx), m_ctx(ctx) {}
13943         void operator()(expr const &e) { return collect(e, false); }
13944     };
13945
13946     void sort_fvars(buffer<expr> &fvars) { ::lean::sort_fvars(m_lctx, fvars); }
13947
13948     /* Initialize `spec_ctx` fields: `m_vars`. */
13949     void specialize_init_deps(expr const &fn, buffer<expr> const &args,
13950                               spec_ctx &ctx) {
13951         lean_assert(is_constant(fn));
13952         buffer<spec_arg_kind> kinds;
13953         get_arg_kinds(const_name(fn), kinds);
13954         bool has_attr = has_specialize_attribute(const_name(fn));
13955         dep_collector collect(m_lctx, ctx);
13956         unsigned sz = std::min(kinds.size(), args.size());
13957         unsigned i = sz;
13958         bool found_inst = false;
13959         while (i > 0) {
13960             --i;
13961             if (is_fvar(args[i])) {
13962                 lean_trace(name({"compiler", "spec_candidate"}),
13963                            local_decl d = m_lctx.get_local_decl(args[i]);
13964                            tout() << "specialize_init_deps [" << i
13965                                   << "]: " << trace_pp_expr(args[i]) << " : "
13966                                   << trace_pp_expr(d.get_type());
13967                            if (auto v = d.get_value()) tout()
13968                            << " := " << trace_pp_expr(*v);
13969                            tout() << "\n";);
13970             }
13971             switch (kinds[i]) {
13972                 case spec_arg_kind::Other:
13973                     break;
13974                 case spec_arg_kind::FixedInst:
13975                     collect(args[i]);
13976                     found_inst = true;
13977                     break;
13978                 case spec_arg_kind::Fixed:
13979                     break;  // We have disabled this kind of argument
13980                 case spec_arg_kind::FixedHO:
13981                 case spec_arg_kind::FixedNeutral:
13982                     if (has_attr || found_inst) {
13983                         collect(args[i]);
13984                     }
13985                     break;
13986             }
13987         }
13988         sort_fvars(ctx.m_vars);
13989         sort_fvars(ctx.m_params);
13990         lean_trace(name({"compiler", "spec_candidate"}),
13991                    tout() << "candidate: " << mk_app(fn, args) << "\nclosure:";
13992                    for (expr const &p
13993                         : ctx.m_vars) tout()
13994                    << " " << trace_pp_expr(p);
13995                    tout() << "\nparams:"; for (expr const &p
13996                                              : ctx.m_params) tout()
13997                                         << " " << trace_pp_expr(p);
13998                    tout() << "\n";);
13999     }
```

```
14000
14001     static bool contains(buffer<optional<expr>> const &mask, expr const &e) {
14002         for (optional<expr> const &o : mask) {
14003             if (o && *o == e) return true;
14004         }
14005         return false;
14006     }
14007
14008     optional<expr> adjust_rec_apps(expr e, buffer<optional<expr>> const &mask,
14009                                    spec_ctx &ctx) {
14010         switch (e.kind()) {
14011             case expr_kind::App:
14012                 if (is_cases_on_app(env(), e)) {
14013                     buffer<expr> args;
14014                     expr const &c = get_app_args(e, args);
14015                     /* visit minor premises */
14016                     unsigned minor_idx;
14017                     unsigned minors_end;
14018                     std::tie(minor_idx, minors_end) =
14019                         get_cases_on_minors_range(env(), const_name(c));
14020                     for (; minor_idx < minors_end; minor_idx++) {
14021                         optional<expr> new_arg =
14022                             adjust_rec_apps(args[minor_idx], mask, ctx);
14023                         if (!new_arg) return none_expr();
14024                         args[minor_idx] = *new_arg;
14025                     }
14026                     return some_expr(mk_app(c, args));
14027                 } else {
14028                     expr const &fn = get_app_fn(e);
14029                     if (!is_constant(fn) || !ctx.in_mutual_decl(const_name(fn)))
14030                         return some_expr(e);
14031                     buffer<expr> args;
14032                     get_app_args(e, args);
14033                     buffer<bool> bmask;
14034                     get_bool_mask(const_name(fn), args.size(), bmask);
14035                     lean_assert(bmask.size() <= args.size());
14036                     buffer<optional<expr>> new_mask;
14037                     bool found = false;
14038                     for (unsigned i = 0; i < bmask.size(); i++) {
14039                         if (bmask[i] && contains(mask, args[i])) {
14040                             found = true;
14041                             new_mask.push_back(some_expr(args[i]));
14042                         } else {
14043                             new_mask.push_back(none_expr());
14044                         }
14045                     }
14046                     if (!found) return some_expr(e);
14047                     optional<name> new_fn_name =
14048                         spec_preprocess(fn, new_mask, ctx);
14049                     if (!new_fn_name) return none_expr();
14050                     expr r = mk_constant(*new_fn_name);
14051                     r = mk_app(r, ctx.m_params);
14052                     for (unsigned i = 0; i < bmask.size(); i++) {
14053                         if (!bmask[i] || !contains(mask, args[i]))
14054                             r = mk_app(r, args[i]);
14055                     }
14056                     for (unsigned i = bmask.size(); i < args.size(); i++) {
14057                         r = mk_app(r, args[i]);
14058                     }
14059                     return some_expr(r);
14060                 }
14061             case expr_kind::Lambda: {
14062                 buffer<expr> entries;
14063                 while (is_lambda(e)) {
14064                     entries.push_back(e);
14065                     e = binding_body(e);
14066                 }
14067                 optional<expr> new_e = adjust_rec_apps(e, mask, ctx);
14068                 if (!new_e) return none_expr();
14069                 expr r = *new_e;
```

```
14070                 unsigned i = entries.size();
14071                 while (i > 0) {
14072                     --i;
14073                     expr l = entries[i];
14074                     r = mk_lambda(binding_name(l), binding_domain(l), r);
14075                 }
14076                 return some_expr(r);
14077             }
14078             case expr_kind::Let: {
14079                 buffer<pair<expr, expr>> entries;
14080                 while (is_let(e)) {
14081                     optional<expr> v = adjust_rec_apps(let_value(e), mask, ctx);
14082                     if (!v) return none_expr();
14083                     expr new_val = *v;
14084                     entries.emplace_back(e, new_val);
14085                     e = let_body(e);
14086                 }
14087                 optional<expr> new_e = adjust_rec_apps(e, mask, ctx);
14088                 if (!new_e) return none_expr();
14089                 expr r = *new_e;
14090                 unsigned i = entries.size();
14091                 while (i > 0) {
14092                     --i;
14093                     expr l = entries[i].first;
14094                     expr v = entries[i].second;
14095                     r = mk_let(let_name(l), let_type(l), v, r);
14096                 }
14097                 return some_expr(r);
14098             }
14099             default:
14100                 return some_expr(e);
14101         }
14102     }
14103
14104     optional<expr> get_code(expr const &fn) {
14105         lean_assert(is_constant(fn));
14106         if (m_to_respecialize.contains(const_name(fn))) {
14107             for (auto const &d : m_new_decls) {
14108                 if (d.fst() == const_name(fn)) return optional<expr>(d.snd());
14109             }
14110         }
14111         optional<constant_info> info =
14112             env().find(mk_cstage1_name(const_name(fn)));
14113         if (!info || !info->is_definition()) return optional<expr>();
14114         return optional<expr>(
14115             instantiate_value_lparams(*info, const_levels(fn)));
14116     }
14117
14118     optional<name> spec_preprocess(expr const &fn,
14119                                    buffer<optional<expr>> const &mask,
14120                                    spec_ctx &ctx) {
14121         lean_assert(is_constant(fn));
14122         lean_assert(ctx.in_mutual_decl(const_name(fn)));
14123         expr key = mk_cache_key(fn, mask);
14124         if (name const *r = ctx.m_cache.find(key)) {
14125             lean_trace(name({"compiler", "specialize"}),
14126                        tout() << "spec_preprocess: " << trace_pp_expr(key)
14127                               << " ==> " << *r << "\n";);
14128             return optional<name>(*r);
14129         }
14130
14131         optional<expr> new_code_opt = get_code(fn);
14132         if (!new_code_opt) return optional<name>();
14133         expr new_code = *new_code_opt;
14134
14135         name new_name = mk_spec_name(const_name(fn));
14136         ctx.m_cache.insert(key, new_name);
14137         lean_trace(
14138             name({"compiler", "specialize"}),
14139             tout() << "spec_preprocess update cache: " << trace_pp_expr(key)
```

```
14140                   << " ===> " << new_name << "\n";);
14141         flet<local_ctx> save_lctx(m_lctx, m_lctx);
14142         buffer<expr> fvars;
14143         buffer<expr> new_fvars;
14144         for (optional<expr> const &b : mask) {
14145             lean_assert(is_lambda(new_code));
14146             if (b) {
14147                 lean_assert(is_fvar(*b));
14148                 fvars.push_back(*b);
14149             } else {
14150                 expr type = instantiate_rev(binding_domain(new_code),
14151                                             fvars.size(), fvars.data());
14152                 expr new_fvar =
14153                     m_lctx.mk_local_decl(ngen(), binding_name(new_code), type,
14154                                          binding_info(new_code));
14155                 new_fvars.push_back(new_fvar);
14156                 fvars.push_back(new_fvar);
14157             }
14158             new_code = binding_body(new_code);
14159         }
14160         new_code = instantiate_rev(new_code, fvars.size(), fvars.data());
14161         lean_trace(name({"compiler", "specialize"}),
14162                    tout() << "before adjust_rec_apps: " << trace_pp_expr(fn)
14163                           << " " << mask.size() << "\n"
14164                           << trace_pp_expr(new_code) << "\n";);
14165         optional<expr> c = adjust_rec_apps(new_code, mask, ctx);
14166         if (!c) return optional<name>();
14167         new_code = *c;
14168         new_code = m_lctx.mk_lambda(new_fvars, new_code);
14169         ctx.m_pre_decls.push_back(comp_decl(new_name, new_code));
14170         // lean_trace(name({"compiler", "spec_info"}), tout() << "new
14171         // specialization " << new_name << " :=\n" << new_code << "\n";);
14172         return optional<name>(new_name);
14173     }
14174
14175     expr eta_expand_specialization(expr e) {
14176         return lcnf_eta_expand(m_st, local_ctx(), e);
14177     }
14178
14179     expr abstract_spec_ctx(spec_ctx const &ctx, expr const &code) {
14180         /* Important: we cannot use
14181            ```
14182            m_lctx.mk_lambda(ctx.m_vars, code)
14183            ```
14184            because we may want to lambda abstract let-variables in `ctx.m_vars`
14185            to avoid code duplication. See comment at `spec_ctx` declaration.
14186
14187            Remark: lambda-abstracting let-decls may introduce type errors
14188            when using dependent types. This is yet another place where
14189            typeability may be lost. */
14190         name_set letvars_in_params;
14191         for (expr const &x : ctx.m_params) {
14192             if (m_lctx.get_local_decl(x).get_value())
14193                 letvars_in_params.insert(fvar_name(x));
14194         }
14195         unsigned n = ctx.m_vars.size();
14196         expr const *fvars = ctx.m_vars.data();
14197         expr r = abstract(code, n, fvars);
14198         unsigned i = n;
14199         while (i > 0) {
14200             --i;
14201             local_decl const &decl = m_lctx.get_local_decl(fvar_name(fvars[i]));
14202             expr type = abstract(decl.get_type(), i, fvars);
14203             optional<expr> val = decl.get_value();
14204             if (val && !letvars_in_params.contains(fvar_name(fvars[i]))) {
14205                 r = ::lean::mk_let(decl.get_user_name(), type,
14206                                    abstract(*val, i, fvars), r);
14207             } else {
14208                 r = ::lean::mk_lambda(decl.get_user_name(), type, r,
14209                                       decl.get_info());
```

```
14210             }
14211         }
14212         return r;
14213     }
14214
14215     optional<comp_decl> mk_new_decl(comp_decl const &pre_decl,
14216                                     buffer<expr> const &fvars,
14217                                     buffer<expr> const &fvar_vals,
14218                                     spec_ctx &ctx) {
14219         lean_assert(fvars.size() == fvar_vals.size());
14220         name n = pre_decl.fst();
14221         expr code = pre_decl.snd();
14222         flet<local_ctx> save_lctx(m_lctx, m_lctx);
14223         /* Add fvars decls */
14224         type_checker tc(m_st, m_lctx);
14225         buffer<expr> new_let_decls;
14226         name y("_y");
14227         for (unsigned i = 0; i < fvars.size(); i++) {
14228             expr type = tc.infer(fvar_vals[i]);
14229             if (is_irrelevant_type(m_st, m_lctx, type)) {
14230                 /* In LCNF, the type `ty` at `let x : ty := v in t` must not be
14231                  * irrelevant. */
14232                 code = replace_fvar(code, fvars[i], fvar_vals[i]);
14233             } else {
14234                 expr new_fvar = m_lctx
14235                                     .mk_local_decl(fvar_name(fvars[i]),
14236                                                    y.append_after(i + 1), type,
14237                                                    fvar_vals[i])
14238                                     .mk_ref();
14239                 new_let_decls.push_back(new_fvar);
14240             }
14241         }
14242         code = m_lctx.mk_lambda(new_let_decls, code);
14243         code = abstract_spec_ctx(ctx, code);
14244         lean_trace(name("compiler", "spec_info"),
14245                    tout() << "specialized code " << n << "\n"
14246                           << trace_pp_expr(code) << "\n";);
14247         lean_assert(!has_fvar(code));
14248         /* We add the auxiliary declaration `n` as a "meta" axiom to the
14249            environment. This is a hack to make sure we can use `csimp` to
14250            simplify `code` and other definitions that use `n`. `csimp` uses the
14251            kernel type checker to infer types, and it will fail to infer the
14252            type of `n`-applications if we do not have an entry in the
14253            environment.
14254
14255            Remark: we mark the axiom as `meta` to make sure it does not polute
14256            the environment regular definitions.
14257
14258            We also considered the following cleaner solution: modify `csimp` to
14259            use a custom type checker that takes the types of auxiliary
14260            declarations such as `n` into account. A custom type checker would be
14261            extra work, but it has other benefits. For example, it could have
14262            better support for type errors introduced by `csimp`. */
14263         try {
14264             expr type = cheap_beta_reduce(type_checker(m_st).infer(code));
14265             declaration aux_ax = mk_axiom(n, names(), type, true /* meta */);
14266             m_st.env() = env().add(aux_ax, false);
14267         } catch (exception &) {
14268             /* We may fail to infer the type of code, since it may be recursive
14269                This is a workaround. When we re-implement the compiler in Lean,
14270                we should write code to infer type that tolerates undefined
14271                constants, *AnyType*, etc.
14272
14273                We just do not specialize when we cannot infer the type. */
14274             return optional<comp_decl>();
14275         }
14276         code = eta_expand_specialization(code);
14277         // lean_trace(name("compiler", "spec_info"), tout() << "STEP 2 " << n <<
14278         // "\n" << code << "\n";);
14279         code = csimp(env(), code, m_cfg);
```

```
14280         code = visit(code);
14281         lean_trace(name("compiler", "specialize"),
14282                    tout() << "new code " << n << "\n"
14283                           << trace_pp_expr(code) << "\n";);
14284         comp_decl new_decl(n, code);
14285         m_new_decls.push_back(new_decl);
14286         return optional<comp_decl>(new_decl);
14287     }
14288
14289     optional<expr> get_closed(expr const &e) {
14290         if (has_univ_param(e)) return none_expr();
14291         switch (e.kind()) {
14292             case expr_kind::MVar:
14293                 lean_unreachable();
14294             case expr_kind::Lit:
14295                 return some_expr(e);
14296             case expr_kind::BVar:
14297                 return some_expr(e);
14298             case expr_kind::Sort:
14299                 return some_expr(e);
14300             case expr_kind::Const:
14301                 return some_expr(e);
14302             case expr_kind::FVar:
14303                 if (auto v = m_lctx.get_local_decl(e).get_value()) {
14304                     return get_closed(*v);
14305                 } else {
14306                     return none_expr();
14307                 }
14308             case expr_kind::MData:
14309                 return get_closed(mdata_expr(e));
14310             case expr_kind::Proj: {
14311                 optional<expr> new_s = get_closed(proj_expr(e));
14312                 if (!new_s) return none_expr();
14313                 return some_expr(update_proj(e, *new_s));
14314             }
14315             case expr_kind::Pi:
14316             case expr_kind::Lambda: {
14317                 optional<expr> dom = get_closed(binding_domain(e));
14318                 if (!dom) return none_expr();
14319                 optional<expr> body = get_closed(binding_body(e));
14320                 if (!body) return none_expr();
14321                 return some_expr(update_binding(e, *dom, *body));
14322             }
14323             case expr_kind::App: {
14324                 buffer<expr> args;
14325                 expr const &fn = get_app_args(e, args);
14326                 optional<expr> new_fn = get_closed(fn);
14327                 if (!new_fn) return none_expr();
14328                 for (expr &arg : args) {
14329                     optional<expr> new_arg = get_closed(arg);
14330                     if (!new_arg) return none_expr();
14331                     arg = *new_arg;
14332                 }
14333                 return some_expr(mk_app(*new_fn, args));
14334             }
14335             case expr_kind::Let: {
14336                 optional<expr> type = get_closed(let_type(e));
14337                 if (!type) return none_expr();
14338                 optional<expr> val = get_closed(let_value(e));
14339                 if (!val) return none_expr();
14340                 optional<expr> body = get_closed(let_body(e));
14341                 if (!body) return none_expr();
14342                 return some_expr(update_let(e, *type, *val, *body));
14343             }
14344         }
14345         lean_unreachable();
14346     }
14347
14348     optional<expr> specialize(expr const &fn, buffer<expr> const &args,
14349                               spec_ctx &ctx) {
```

```
14350         if (!is_specialize_candidate(fn, args)) return none_expr();
14351         // lean_trace(name("compiler", "specialize"), tout() << "specialize: "
14352         // << fn << "\n";);
14353         bool has_attr = has_specialize_attribute(const_name(fn));
14354         specialize_init_deps(fn, args, ctx);
14355         buffer<bool> bmask;
14356         get_bool_mask(const_name(fn), args.size(), bmask);
14357         buffer<optional<expr>> mask;
14358         buffer<expr> fvars;
14359         buffer<expr> fvar_vals;
14360         bool gcache_enabled = true;
14361         buffer<expr> gcache_key_args;
14362         for (unsigned i = 0; i < bmask.size(); i++) {
14363             if (bmask[i]) {
14364                 if (gcache_enabled) {
14365                     if (optional<expr> c = get_closed(args[i])) {
14366                         gcache_key_args.push_back(*c);
14367                     } else {
14368                         /* We only cache specialization results if arguments
14369                          * (expanded by the specializer) are closed. */
14370                         gcache_enabled = false;
14371                     }
14372                 }
14373                 name n = ngen().next();
14374                 expr fvar = mk_fvar(n);
14375                 fvars.push_back(fvar);
14376                 fvar_vals.push_back(args[i]);
14377                 mask.push_back(some_expr(fvar));
14378             } else {
14379                 mask.push_back(none_expr());
14380                 if (gcache_enabled) gcache_key_args.push_back(expr());
14381             }
14382         }
14383
14384         // We try to respecialize if the current application is over-applied,
14385         // and it has additional lambda as arguments.
14386         bool respecialize = false;
14387         for (unsigned i = mask.size(); i < args.size(); i++) {
14388             expr w = find(args[i]);
14389             if (is_lambda(w) || is_constant(get_app_fn(w))) {
14390                 respecialize = true;
14391                 break;
14392             }
14393         }
14394
14395         optional<name> new_fn_name;
14396         expr key;
14397         /* When `m_params.size > 1`, it is not safe to reuse cached
14398            specialization. See test `tests/lean/run/specbug.lean`. This is a bit
14399            hackish, but should not produce increase the generated code size too
14400            much. On Dec 20 2020, before this fix, 5246 specializations were
14401            reused, but only 11 had `m_params.size > 1`. This file will be
14402            deleted. So, it is not worth designing a better caching scheme.
14403            TODO: when we reimplement this module in Lean, we should have a
14404            better caching heuristic. */
14405         if (gcache_enabled && ctx.m_params.size() <= 1) {
14406             key = mk_app(fn, gcache_key_args);
14407             if (optional<name> it = get_cached_specialization(env(), key)) {
14408                 lean_trace(
14409                     name({"compiler", "specialize"}),
14410                     tout() << "get_cached_specialization ["
14411                            << ctx.m_params.size() << "]: " << *it << "\n";
14412                     unsigned i = 0;
14413                     type_checker tc(m_st, m_lctx); for (expr const &x
14414                                                         : ctx.m_params) {
14415                         tout() << ">> [" << i
14416                                << "]: " << trace_pp_expr(tc.infer(x)) << "\n";
14417                         i++;
14418                     } tout() << trace_pp_expr(key)
14419                              << "\n";);
```

```
14420                 // std::cerr << *it << " " << ctx.m_vars.size() << " " <<
14421                 // ctx.m_params.size() << "\n";
14422                 new_fn_name = *it;
14423             }
14424         }
14425         if (!new_fn_name) {
14426             /* Cache does not contain specialization result */
14427             new_fn_name = spec_preprocess(fn, mask, ctx);
14428             if (!new_fn_name) return none_expr();
14429             buffer<comp_decl> new_decls;
14430             for (comp_decl const &pre_decl : ctx.m_pre_decls) {
14431                 if (auto new_decl_opt =
14432                         mk_new_decl(pre_decl, fvars, fvar_vals, ctx)) {
14433                     new_decls.push_back(*new_decl_opt);
14434                 } else {
14435                     return none_expr();
14436                 }
14437             }
14438             /* We should only re-specialize if the original function was marked
14439                with `[specialize]` attribute. Recall that we always specialize
14440                functions containing instance implicit arguments. This is a
14441                temporary workaround until we implement a proper code
14442                specializer.
14443             */
14444             if (has_attr && respecialize) {
14445                 for (comp_decl const &new_decl : new_decls) {
14446                     m_to_respecialize.insert(new_decl.fst());
14447                 }
14448                 m_st.env() = update_spec_info(env(), new_decls);
14449             }
14450             if (gcache_enabled) {
14451                 lean_trace(
14452                     name({"compiler", "specialize"}),
14453                     tout() << "get_cached_specialization ["
14454                            << ctx.m_params.size() << "] UPDATE " << *new_fn_name
14455                            << "\n";
14456                     unsigned i = 0;
14457                     type_checker tc(m_st, m_lctx); for (expr const &x
14458                                                         : ctx.m_params) {
14459                         tout() << ">> [" << i
14460                                << "]: " << trace_pp_expr(tc.infer(x)) << "\n";
14461                         i++;
14462                     } tout() << trace_pp_expr(key)
14463                              << "\n";);
14464                 m_st.env() = cache_specialization(env(), key, *new_fn_name);
14465             }
14466         }
14467         expr r = mk_constant(*new_fn_name);
14468         r = mk_app(r, ctx.m_params);
14469         for (unsigned i = 0; i < bmask.size(); i++) {
14470             if (!bmask[i]) r = mk_app(r, args[i]);
14471         }
14472         for (unsigned i = bmask.size(); i < args.size(); i++) {
14473             r = mk_app(r, args[i]);
14474         }
14475         return some_expr(r);
14476     }
14477
14478     expr visit_app(expr const &e) {
14479         if (is_cases_on_app(env(), e)) {
14480             return visit_cases_on(e);
14481         } else {
14482             buffer<expr> args;
14483             expr fn = get_app_args(e, args);
14484             if (!is_constant(fn) ||
14485                 has_nospecialize_attribute(env(), const_name(fn)) ||
14486                 (is_instance(env(), const_name(fn)) &&
14487                  !has_specialize_attribute(const_name(fn)))) {
14488                 return e;
14489             }
```

```
14490             optional<spec_info> info =
14491                 get_specialization_info(env(), const_name(fn));
14492             if (!info) return e;
14493             spec_ctx ctx;
14494             ctx.m_mutual = info->get_mutual_decls();
14495             if (optional<expr> r = specialize(fn, args, ctx)) {
14496                 if (m_to_respecialize.contains(const_name(get_app_fn(*r))))
14497                     return visit(*r);
14498                 else
14499                     return *r;
14500             } else {
14501                 return e;
14502             }
14503         }
14504     }
14505
14506     expr visit(expr const &e) {
14507         switch (e.kind()) {
14508             case expr_kind::App:
14509                 return visit_app(e);
14510             case expr_kind::Lambda:
14511                 return visit_lambda(e);
14512             case expr_kind::Let:
14513                 return visit_let(e);
14514             default:
14515                 return e;
14516         }
14517     }
14518
14519    public:
14520     specialize_fn(environment const &env, csimp_cfg const &cfg)
14521         : m_st(env), m_cfg(cfg), m_at("_at"), m_spec("_spec") {}
14522
14523     pair<environment, comp_decls> operator()(comp_decl const &d) {
14524         m_base_name = d.fst();
14525         lean_trace(name({"compiler", "specialize"}),
14526                    tout() << "INPUT: " << d.fst() << "\n"
14527                           << trace_pp_expr(d.snd()) << "\n";);
14528         expr new_v = visit(d.snd());
14529         comp_decl new_d(d.fst(), new_v);
14530         return mk_pair(env(),
14531                        append(comp_decls(m_new_decls), comp_decls(new_d)));
14532     }
14533 };
14534
14535 pair<environment, comp_decls> specialize_core(environment const &env,
14536                                                comp_decl const &d,
14537                                                csimp_cfg const &cfg) {
14538     return specialize_fn(env, cfg)(d);
14539 }
14540
14541 pair<environment, comp_decls> specialize(environment env, comp_decls const &ds,
14542                                          csimp_cfg const &cfg) {
14543     env = update_spec_info(env, ds);
14544     comp_decls r;
14545     for (comp_decl const &d : ds) {
14546         comp_decls new_ds;
14547         if (has_specialize_attribute(env, d.fst())) {
14548             r = append(r, comp_decls(d));
14549         } else {
14550             std::tie(env, new_ds) = specialize_core(env, d, cfg);
14551             r = append(r, new_ds);
14552         }
14553     }
14554     return mk_pair(env, r);
14555 }
14556
14557 void initialize_specialize() {
14558     register_trace_class({"compiler", "spec_info"});
14559     register_trace_class({"compiler", "spec_candidate"});
```

```
14560 }
14561
14562 void finalize_specialize() {}
14563 }  // namespace lean
14564 // ::::::::::::::
14565 // compiler/struct_cases_on.cpp
14566 // ::::::::::::::
14567 /*
14568 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
14569 Released under Apache 2.0 license as described in the file LICENSE.
14570
14571 Author: Leonardo de Moura
14572 */
14573 #include <lean/flet.h>
14574
14575 #include "kernel/abstract.h"
14576 #include "kernel/instantiate.h"
14577 #include "kernel/type_checker.h"
14578 #include "library/compiler/util.h"
14579 #include "library/suffixes.h"
14580 #include "library/trace.h"
14581
14582 namespace lean {
14583 class struct_cases_on_fn {
14584     type_checker::state m_st;
14585     local_ctx m_lctx;
14586     name_set m_scrutinies; /* Set of variables `x` such that there is `casesOn x
14587                               ...` in the context */
14588     name_map<name> m_first_proj; /* Map from variable `x` to the first
14589                                     projection `y := x.i` in the context */
14590     name_set m_updated; /* Set of variables `x` such that there is a `S.mk ...
14591                            x.i ... */
14592     name m_fld{"_d"};
14593     unsigned m_next_idx{1};
14594
14595     environment const &env() { return m_st.env(); }
14596
14597     name_generator &ngen() { return m_st.ngen(); }
14598
14599     name next_field_name() {
14600         name r = m_fld.append_after(m_next_idx);
14601         m_next_idx++;
14602         return r;
14603     }
14604
14605     expr find(expr const &e) const {
14606         if (is_fvar(e)) {
14607             if (optional<local_decl> decl = m_lctx.find_local_decl(e)) {
14608                 if (optional<expr> v = decl->get_value()) {
14609                     if (!is_join_point_name(decl->get_user_name()))
14610                         return find(*v);
14611                 }
14612             }
14613         } else if (is_mdata(e)) {
14614             return find(mdata_expr(e));
14615         }
14616         return e;
14617     }
14618
14619     expr visit_cases(expr const &e) {
14620         flet<name_set> save(m_scrutinies, m_scrutinies);
14621         buffer<expr> args;
14622         expr const &c = get_app_args(e, args);
14623         expr const &major = args[0];
14624         if (is_fvar(major)) m_scrutinies.insert(fvar_name(major));
14625         for (unsigned i = 1; i < args.size(); i++) {
14626             args[i] = visit(args[i]);
14627         }
14628         return mk_app(c, args);
14629     }
```

```
14630
14631     expr visit_app(expr const &e) {
14632         if (is_cases_on_app(env(), e)) {
14633             return visit_cases(e);
14634         } else if (is_constructor_app(env(), e)) {
14635             buffer<expr> args;
14636             expr const &k = get_app_args(e, args);
14637             lean_assert(is_constant(k));
14638             constructor_val k_val =
14639                 env().get(const_name(k)).to_constructor_val();
14640             for (unsigned i = k_val.get_nparams(), idx = 0; i < args.size();
14641                  i++, idx++) {
14642                 expr arg = find(args[i]);
14643                 if (is_proj(arg) && proj_idx(arg) == idx &&
14644                     is_fvar(proj_expr(arg))) {
14645                     m_updated.insert(fvar_name(proj_expr(arg)));
14646                 }
14647             }
14648             return e;
14649         } else {
14650             return e;
14651         }
14652     }
14653
14654     expr visit_lambda(expr e) {
14655         buffer<expr> fvars;
14656         while (is_lambda(e)) {
14657             lean_assert(!has_loose_bvars(binding_domain(e)));
14658             expr new_fvar = m_lctx.mk_local_decl(
14659                 ngen(), binding_name(e), binding_domain(e), binding_info(e));
14660             fvars.push_back(new_fvar);
14661             e = binding_body(e);
14662         }
14663         e = instantiate_rev(e, fvars.size(), fvars.data());
14664         e = visit(e);
14665         return m_lctx.mk_lambda(fvars, e);
14666     }
14667
14668     /* Return `some s` if `rhs` is of the form `s.i`, and `s` is a free
14669        variables that has not been
14670        scrutinized yet, and `s.i` is the first time it is being projected. */
14671     optional<name> is_candidate(expr const &rhs) {
14672         if (!is_proj(rhs)) return optional<name>();
14673         expr const &s = proj_expr(rhs);
14674         if (!is_fvar(s)) return optional<name>();
14675         name const &s_name = fvar_name(s);
14676         if (m_scrutinies.contains(s_name)) return optional<name>();
14677         if (m_first_proj.contains(s_name)) return optional<name>();
14678         return optional<name>(s_name);
14679     }
14680
14681     static void get_struct_field_types(type_checker::state &st,
14682                                        name const &S_name,
14683                                        buffer<expr> &result) {
14684         environment const &env = st.env();
14685         constant_info info = env.get(S_name);
14686         lean_assert(info.is_inductive());
14687         inductive_val I_val = info.to_inductive_val();
14688         lean_assert(length(I_val.get_cnstrs()) == 1);
14689         constant_info ctor_info = env.get(head(I_val.get_cnstrs()));
14690         expr type = ctor_info.get_type();
14691         unsigned nparams = I_val.get_nparams();
14692         local_ctx lctx;
14693         buffer<expr> telescope;
14694         to_telescope(env, lctx, st.ngen(), type, telescope);
14695         lean_assert(telescope.size() >= nparams);
14696         for (unsigned i = nparams; i < telescope.size(); i++) {
14697             expr ftype = lctx.get_type(telescope[i]);
14698             if (is_irrelevant_type(st, lctx, ftype)) {
14699                 result.push_back(mk_enf_neutral_type());
```

```
14700             } else {
14701                 type_checker tc(st, lctx);
14702                 ftype = tc.whnf(ftype);
14703                 if (is_usize_type(ftype)) {
14704                     result.push_back(ftype);
14705                 } else if (is_builtin_scalar(ftype)) {
14706                     result.push_back(ftype);
14707                 } else if (optional<unsigned> sz = is_enum_type(env, ftype)) {
14708                     optional<expr> uint = to_uint_type(*sz);
14709                     if (!uint)
14710                         throw exception(
14711                             "code generation failed, enumeration type is too "
14712                             "big");
14713                     result.push_back(*uint);
14714                 } else {
14715                     result.push_back(mk_enf_object_type());
14716                 }
14717             }
14718         }
14719     }
14720
14721     bool should_add_cases_on(local_decl const &decl) {
14722         expr val = *decl.get_value();
14723         if (!is_proj(val)) return false;
14724         expr const &s = proj_expr(val);
14725         if (!is_fvar(s) || !m_updated.contains(fvar_name(s))) return false;
14726         name const *x = m_first_proj.find(fvar_name(s));
14727         return x && *x == decl.get_name();
14728     }
14729
14730     expr visit_let(expr e) {
14731         flet<name_map<name>> save(m_first_proj, m_first_proj);
14732         buffer<expr> fvars;
14733         while (is_let(e)) {
14734             lean_assert(!has_loose_bvars(let_type(e)));
14735             expr type = let_type(e);
14736             expr val = visit(
14737                 instantiate_rev(let_value(e), fvars.size(), fvars.data()));
14738             name n = let_name(e);
14739             e = let_body(e);
14740             expr new_fvar = m_lctx.mk_local_decl(ngen(), n, type, val);
14741             fvars.push_back(new_fvar);
14742             if (optional<name> s = is_candidate(val)) {
14743                 m_first_proj.insert(*s, fvar_name(new_fvar));
14744             }
14745         }
14746         e = visit(instantiate_rev(e, fvars.size(), fvars.data()));
14747         e = abstract(e, fvars.size(), fvars.data());
14748         unsigned i = fvars.size();
14749         while (i > 0) {
14750             --i;
14751             expr const &x = fvars[i];
14752             lean_assert(is_fvar(x));
14753             local_decl decl = m_lctx.get_local_decl(x);
14754             expr type = decl.get_type();
14755             expr val = *decl.get_value();
14756             expr aval = abstract(val, i, fvars.data());
14757             e = mk_let(decl.get_user_name(), type, aval, e);
14758             if (should_add_cases_on(decl)) {
14759                 lean_assert(is_proj(val));
14760                 expr major = proj_expr(aval);
14761                 buffer<expr> field_types;
14762                 get_struct_field_types(m_st, proj_sname(val), field_types);
14763                 e = lift_loose_bvars(e, field_types.size());
14764                 unsigned i = field_types.size();
14765                 while (i > 0) {
14766                     --i;
14767                     e = mk_lambda(next_field_name(), field_types[i], e);
14768                 }
14769                 e = mk_app(mk_constant(name(proj_sname(val), g_cases_on)),
```

```
                            major, e);
            }
        }
        return e;
    }

    expr visit(expr const &e) {
        switch (e.kind()) {
            case expr_kind::App:
                return visit_app(e);
            case expr_kind::Lambda:
                return visit_lambda(e);
            case expr_kind::Let:
                return visit_let(e);
            default:
                return e;
        }
    }

   public:
    struct_cases_on_fn(environment const &env) : m_st(env) {}

    expr operator()(expr const &e) { return visit(e); }
};

expr struct_cases_on(environment const &env, expr const &e) {
    return struct_cases_on_fn(env)(e);
}
}  // namespace lean
// ::::::::::::::
// compiler/util.cpp
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#include <algorithm>
#include <cctype>
#include <limits>
#include <string>
#include <unordered_set>
#include <vector>

#include "kernel/for_each_fn.h"
#include "kernel/instantiate.h"
#include "kernel/kernel_exception.h"
#include "kernel/replace_fn.h"
#include "kernel/type_checker.h"
#include "library/aux_recursors.h"
#include "library/compiler/eager_lambda_lifting.h"
#include "library/compiler/lambda_lifting.h"
#include "library/compiler/util.h"
#include "library/constants.h"
#include "library/replace_visitor.h"
#include "library/suffixes.h"
#include "library/trace.h"
#include "library/util.h"
#include "util/name_hash_set.h"

namespace lean {
optional<unsigned> is_enum_type(environment const &env, name const &I) {
    constant_info info = env.get(I);
    if (!info.is_inductive()) return optional<unsigned>();
    /* `decidable` is morally an enumeration type */
    if (I == get_decidable_name()) return optional<unsigned>(1);
    unsigned n = 0;
    names cs = info.to_inductive_val().get_cnstrs();
    if (length(cs) == 1) {
```

```
14840         /* We do not consider types such as `unit` as enumeration types.
14841            There is no motivation for them to be, since nobody will use them in
14842            composite datastructures. So, we don't save space, but we keep
14843            boxing/unboxing. Moreover `unit` is used to encode `thunks` which get
14844            closures. Thus, if we treat `unit` as an enumeration type, we will
14845            perform a useless unboxing whenever we force a thunk. */
14846         return optional<unsigned>();
14847     }
14848     for (name const &c : cs) {
14849         if (is_pi(env.get(c).get_type())) return optional<unsigned>();
14850         if (n == std::numeric_limits<unsigned>::max())
14851             return optional<unsigned>();
14852         n++;
14853     }
14854     if (n < (1u << 8)) {
14855         return optional<unsigned>(1);
14856     } else if (n < (1u << 16)) {
14857         return optional<unsigned>(2);
14858     } else {
14859         return optional<unsigned>(4);
14860     }
14861 }
14862
14863 static expr *g_usize = nullptr;
14864 static expr *g_uint8 = nullptr;
14865 static expr *g_uint16 = nullptr;
14866 static expr *g_uint32 = nullptr;
14867 static expr *g_uint64 = nullptr;
14868
14869 optional<expr> to_uint_type(unsigned nbytes) {
14870     /* Remark: we use 0 to denote the size of the type `usize` since it is
14871        platform specific, and we don't want the generated code to be platform
14872        specific. `usize` is 4 in 32-bit machines and 8 in 64-bit. */
14873     switch (nbytes) {
14874         case 0:
14875             return some_expr(*g_usize);
14876         case 1:
14877             return some_expr(*g_uint8);
14878         case 2:
14879             return some_expr(*g_uint16);
14880         case 4:
14881             return some_expr(*g_uint32);
14882         case 8:
14883             return some_expr(*g_uint64);
14884         default:
14885             return none_expr();
14886     }
14887 }
14888
14889 unsigned get_num_nested_lambdas(expr e) {
14890     unsigned r = 0;
14891     while (is_lambda(e)) {
14892         r++;
14893         e = binding_body(e);
14894     }
14895     return r;
14896 }
14897
14898 extern "C" uint8 lean_has_inline_attribute(object *env, object *n);
14899 extern "C" uint8 lean_has_inline_if_reduce_attribute(object *env, object *n);
14900 extern "C" uint8 lean_has_macro_inline_attribute(object *env, object *n);
14901 extern "C" uint8 lean_has_noinline_attribute(object *env, object *n);
14902
14903 bool has_inline_attribute(environment const &env, name const &n) {
14904     return lean_has_inline_attribute(env.to_obj_arg(), n.to_obj_arg());
14905 }
14906 bool has_inline_if_reduce_attribute(environment const &env, name const &n) {
14907     return lean_has_inline_if_reduce_attribute(env.to_obj_arg(),
14908                                                n.to_obj_arg());
14909 }
```

```
14910 bool has_macro_inline_attribute(environment const &env, name const &n) {
14911     return lean_has_macro_inline_attribute(env.to_obj_arg(), n.to_obj_arg());
14912 }
14913 bool has_noinline_attribute(environment const &env, name const &n) {
14914     return lean_has_noinline_attribute(env.to_obj_arg(), n.to_obj_arg());
14915 }
14916
14917 extern "C" uint8 lean_has_never_extract_attribute(object *env, object *n);
14918 bool has_never_extract_attribute(environment const &env, name const &n) {
14919     return lean_has_never_extract_attribute(env.to_obj_arg(), n.to_obj_arg());
14920 }
14921
14922 bool is_lcnf_atom(expr const &e) {
14923     switch (e.kind()) {
14924         case expr_kind::FVar:
14925         case expr_kind::Const:
14926         case expr_kind::Lit:
14927             return true;
14928         default:
14929             return false;
14930     }
14931 }
14932
14933 class elim_trivial_let_decls_fn : public replace_visitor {
14934     virtual expr visit_let(expr const &e) override {
14935         if (is_lcnf_atom(let_value(e))) {
14936             return visit(instantiate(let_body(e), let_value(e)));
14937         } else {
14938             return replace_visitor::visit_let(e);
14939         }
14940     }
14941 };
14942
14943 expr elim_trivial_let_decls(expr const &e) {
14944     return elim_trivial_let_decls_fn()(e);
14945 }
14946
14947 struct unfold_macro_defs_fn : public replace_visitor {
14948     environment const &m_env;
14949     unfold_macro_defs_fn(environment const &env) : m_env(env) {}
14950
14951     virtual expr visit_app(expr const &e) override {
14952         buffer<expr> args;
14953         expr const &fn = get_app_args(e, args);
14954         bool modified = false;
14955         for (expr &arg : args) {
14956             expr new_arg = visit(arg);
14957             if (!is_eqp(new_arg, arg)) modified = true;
14958             arg = new_arg;
14959         }
14960         if (is_constant(fn)) {
14961             name const &n = const_name(fn);
14962             if (has_macro_inline_attribute(m_env, n)) {
14963                 expr new_fn =
14964                     instantiate_value_lparams(m_env.get(n), const_levels(fn));
14965                 std::reverse(args.begin(), args.end());
14966                 return visit(apply_beta(new_fn, args.size(), args.data()));
14967             }
14968         }
14969         expr new_fn = visit(fn);
14970         if (!modified && is_eqp(new_fn, fn))
14971             return e;
14972         else
14973             return mk_app(new_fn, args);
14974     }
14975
14976     virtual expr visit_constant(expr const &e) override {
14977         name const &n = const_name(e);
14978         if (has_macro_inline_attribute(m_env, n)) {
14979             return visit(
```

```
14980                 instantiate_value_lparams(m_env.get(n), const_levels(e)));
14981         } else {
14982             return e;
14983         }
14984     }
14985 };
14986
14987 expr unfold_macro_defs(environment const &env, expr const &e) {
14988     return unfold_macro_defs_fn(env)(e);
14989 }
14990
14991 bool is_cases_on_recursor(environment const &env, name const &n) {
14992     return ::lean::is_aux_recursor(env, n) && n.get_string() == g_cases_on;
14993 }
14994
14995 unsigned get_cases_on_arity(environment const &env, name const &c,
14996                             bool before_erasure) {
14997     lean_assert(is_cases_on_recursor(env, c));
14998     inductive_val I_val = get_cases_on_inductive_val(env, c);
14999     unsigned nminors = I_val.get_ncnstrs();
15000     if (before_erasure) {
15001         unsigned nparams = I_val.get_nparams();
15002         unsigned nindices = I_val.get_nindices();
15003         return nparams + 1 /* motive */ + nindices + 1 /* major */ + nminors;
15004     } else {
15005         return 1 /* major */ + nminors;
15006     }
15007 }
15008
15009 unsigned get_cases_on_major_idx(environment const &env, name const &c,
15010                                 bool before_erasure) {
15011     if (before_erasure) {
15012         inductive_val I_val = get_cases_on_inductive_val(env, c);
15013         return I_val.get_nparams() + 1 /* motive */ + I_val.get_nindices();
15014     } else {
15015         return 0;
15016     }
15017 }
15018
15019 expr get_cases_on_app_major(environment const &env, expr const &c,
15020                             bool before_erasure) {
15021     lean_assert(is_cases_on_app(env, c));
15022     buffer<expr> args;
15023     expr const &fn = get_app_args(c, args);
15024     return args[get_cases_on_major_idx(env, const_name(fn), before_erasure)];
15025 }
15026
15027 pair<unsigned, unsigned> get_cases_on_minors_range(environment const &env,
15028                                                    name const &c,
15029                                                    bool before_erasure) {
15030     inductive_val I_val = get_cases_on_inductive_val(env, c);
15031     unsigned nminors = I_val.get_ncnstrs();
15032     if (before_erasure) {
15033         unsigned nparams = I_val.get_nparams();
15034         unsigned nindices = I_val.get_nindices();
15035         unsigned first_minor_idx =
15036             nparams + 1 /*motive*/ + nindices + 1 /* major */;
15037         return mk_pair(first_minor_idx, first_minor_idx + nminors);
15038     } else {
15039         return mk_pair(1, 1 + nminors);
15040     }
15041 }
15042
15043 expr mk_lc_unreachable(type_checker::state &s, local_ctx const &lctx,
15044                        expr const &type) {
15045     type_checker tc(s, lctx);
15046     expr t = cheap_beta_reduce(type);
15047     level lvl = sort_level(tc.ensure_type(t));
15048     return mk_app(mk_constant(get_lc_unreachable_name(), {lvl}), t);
15049 }
```

```
15050
15051 bool is_join_point_name(name const &n) {
15052     return !n.is_atomic() && n.is_string() &&
15053            strncmp(n.get_string().data(), "_join", 5) == 0;
15054 }
15055
15056 bool is_pseudo_do_join_point_name(name const &n) {
15057     return !n.is_atomic() && n.is_string() &&
15058            strncmp(n.get_string().data(), "_do_jp", 6) == 0;
15059 }
15060
15061 bool has_fvar(expr const &e, expr const &fvar) {
15062     if (!has_fvar(e)) return false;
15063     bool found = false;
15064     for_each(e, [&](expr const &e, unsigned) {
15065         if (!has_fvar(e)) return false;
15066         if (found) return false;
15067         if (is_fvar(e) && fvar_name(fvar) == fvar_name(e)) found = true;
15068         return true;
15069     });
15070     return found;
15071 }
15072
15073 void mark_used_fvars(expr const &e, buffer<expr> const &fvars,
15074                      buffer<bool> &used) {
15075     used.resize(fvars.size(), false);
15076     if (!has_fvar(e) || fvars.empty()) return;
15077     bool all_used = false;
15078     for_each(e, [&](expr const &e, unsigned) {
15079         if (!has_fvar(e)) return false;
15080         if (all_used) return false;
15081         if (is_fvar(e)) {
15082             all_used = true;
15083             for (unsigned i = 0; i < fvars.size(); i++) {
15084                 if (!used[i]) {
15085                     all_used = false;
15086                     if (fvar_name(fvars[i]) == fvar_name(e)) {
15087                         used[i] = true;
15088                         break;
15089                     }
15090                 }
15091             }
15092         }
15093         return true;
15094     });
15095 }
15096
15097 expr replace_fvar(expr const &e, expr const &fvar, expr const &new_term) {
15098     if (!has_fvar(e)) return e;
15099     return replace(e, [&](expr const &e, unsigned) {
15100         if (!has_fvar(e)) return some_expr(e);
15101         if (is_fvar(e) && fvar_name(fvar) == fvar_name(e))
15102             return some_expr(new_term);
15103         return none_expr();
15104     });
15105 }
15106
15107 void sort_fvars(local_ctx const &lctx, buffer<expr> &fvars) {
15108     std::sort(fvars.begin(), fvars.end(), [&](expr const &x, expr const &y) {
15109         return lctx.get_local_decl(x).get_idx() <
15110                lctx.get_local_decl(y).get_idx();
15111     });
15112 }
15113
15114 unsigned get_lcnf_size(environment const &env, expr e) {
15115     unsigned r = 0;
15116     switch (e.kind()) {
15117         case expr_kind::BVar:
15118         case expr_kind::MVar:
15119         case expr_kind::Sort:
```

```
15120         case expr_kind::Lit:
15121         case expr_kind::FVar:
15122         case expr_kind::Pi:
15123         case expr_kind::Proj:
15124         case expr_kind::MData:
15125             return 1;
15126         case expr_kind::Const:
15127             return 1;
15128         case expr_kind::Lambda:
15129             while (is_lambda(e)) {
15130                 e = binding_body(e);
15131             }
15132             return get_lcnf_size(env, e);
15133         case expr_kind::App:
15134             if (is_cases_on_app(env, e)) {
15135                 expr const &c_fn = get_app_fn(e);
15136                 inductive_val I_val =
15137                     env.get(const_name(c_fn).get_prefix()).to_inductive_val();
15138                 unsigned nminors = I_val.get_ncnstrs();
15139                 r = 1;
15140                 for (unsigned i = 0; i < nminors; i++) {
15141                     lean_assert(is_app(e));
15142                     r += get_lcnf_size(env, app_arg(e));
15143                     e = app_fn(e);
15144                 }
15145                 return r;
15146             } else {
15147                 return 1;
15148             }
15149         case expr_kind::Let:
15150             while (is_let(e)) {
15151                 r += get_lcnf_size(env, let_value(e));
15152                 e = let_body(e);
15153             }
15154             return r + get_lcnf_size(env, e);
15155     }
15156     lean_unreachable();
15157 }
15158
15159 static expr *g_neutral_expr = nullptr;
15160 static expr *g_unreachable_expr = nullptr;
15161 static expr *g_object_type = nullptr;
15162 static expr *g_void_type = nullptr;
15163
15164 expr mk_enf_unreachable() { return *g_unreachable_expr; }
15165
15166 expr mk_enf_neutral() { return *g_neutral_expr; }
15167
15168 expr mk_enf_object_type() { return *g_object_type; }
15169
15170 expr mk_llnf_void_type() { return *g_void_type; }
15171
15172 expr mk_enf_neutral_type() { return *g_neutral_expr; }
15173
15174 bool is_enf_neutral(expr const &e) { return e == *g_neutral_expr; }
15175
15176 bool is_enf_unreachable(expr const &e) { return e == *g_unreachable_expr; }
15177
15178 bool is_enf_object_type(expr const &e) { return e == *g_object_type; }
15179
15180 bool is_llnf_void_type(expr const &e) { return e == *g_void_type; }
15181
15182 bool is_runtime_builtin_type(name const &n) {
15183     /* TODO(Leo): use an attribute? */
15184     return n == get_string_name() || n == get_uint8_name() ||
15185            n == get_uint16_name() || n == get_uint32_name() ||
15186            n == get_uint64_name() || n == get_usize_name() ||
15187            n == get_float_name() || n == get_thunk_name() ||
15188            n == get_task_name() || n == get_array_name() ||
15189            n == get_mut_quot_name() || n == get_byte_array_name() ||
```

```
15190            n == get_float_array_name() || n == get_nat_name() ||
15191            n == get_int_name();
15192 }
15193
15194 bool is_runtime_scalar_type(name const &n) {
15195     return n == get_uint8_name() || n == get_uint16_name() ||
15196            n == get_uint32_name() || n == get_uint64_name() ||
15197            n == get_usize_name() || n == get_float_name();
15198 }
15199
15200 bool is_llnf_unboxed_type(expr const &type) {
15201     return type != mk_enf_object_type() && type != mk_enf_neutral_type() &&
15202            !is_pi(type);
15203 }
15204
15205 bool is_irrelevant_type(type_checker::state &st, local_ctx lctx,
15206                         expr const &type) {
15207     if (is_sort(type) || type_checker(st, lctx).is_prop(type)) return true;
15208     expr type_it = type;
15209     if (is_pi(type_it)) {
15210         while (is_pi(type_it)) {
15211             expr fvar = lctx.mk_local_decl(st.ngen(), binding_name(type_it),
15212                                            binding_domain(type_it));
15213             type_it = type_checker(st, lctx).whnf(
15214                 instantiate(binding_body(type_it), fvar));
15215         }
15216         if (is_sort(type_it)) return true;
15217     }
15218     return false;
15219 }
15220
15221 bool is_irrelevant_type(environment const &env, expr const &type) {
15222     type_checker::state st(env);
15223     return is_irrelevant_type(st, local_ctx(), type);
15224 }
15225
15226 void collect_used(expr const &e, name_hash_set &S) {
15227     if (!has_fvar(e)) return;
15228     for_each(e, [&](expr const &e, unsigned) {
15229         if (!has_fvar(e)) return false;
15230         if (is_fvar(e)) {
15231             S.insert(fvar_name(e));
15232             return false;
15233         }
15234         return true;
15235     });
15236 }
15237
15238 bool depends_on(expr const &e, name_hash_set const &s) {
15239     if (!has_fvar(e)) return false;
15240     bool found = false;
15241     for_each(e, [&](expr const &e, unsigned) {
15242         if (!has_fvar(e)) return false;
15243         if (found) return false;
15244         if (is_fvar(e) && s.find(fvar_name(e)) != s.end()) {
15245             found = true;
15246         }
15247         return true;
15248     });
15249     return found;
15250 }
15251
15252 optional<unsigned> has_trivial_structure(environment const &env,
15253                                          name const &I_name) {
15254     if (is_runtime_builtin_type(I_name)) return optional<unsigned>();
15255     inductive_val I_val = env.get(I_name).to_inductive_val();
15256     if (I_val.is_unsafe()) return optional<unsigned>();
15257     if (I_val.get_ncnstrs() != 1) return optional<unsigned>();
15258     buffer<bool> rel_fields;
15259     get_constructor_relevant_fields(env, head(I_val.get_cnstrs()), rel_fields);
```

```
/* The following #pragma is to disable a bogus g++ 4.9 warning at
 * `optional<unsigned> r` */
#if defined(__GNUC__) && !defined(__CLANG__)
#pragma GCC diagnostic ignored "-Wmaybe-uninitialized"
#endif
    optional<unsigned> result;
    for (unsigned i = 0; i < rel_fields.size(); i++) {
        if (rel_fields[i]) {
            if (result) return optional<unsigned>();
            result = i;
        }
    }
    return result;
}

expr mk_runtime_type(type_checker::state &st, local_ctx const &lctx, expr e) {
    try {
        type_checker tc(st, lctx);
        e = tc.whnf(e);

        if (is_constant(e)) {
            name const &c = const_name(e);
            if (is_runtime_scalar_type(c)) {
                return e;
            } else if (c == get_char_name()) {
                return mk_constant(get_uint32_name());
            } else if (c == get_usize_name()) {
                return e;
            } else if (c == get_float_name()) {
                return e;
            } else if (optional<unsigned> nbytes = is_enum_type(st.env(), c)) {
                return *to_uint_type(*nbytes);
            }
        }

        if (is_app_of(e, get_decidable_name())) {
            /* Recall that `decidable A` and `bool` have the same runtime
             * representation. */
            return *to_uint_type(1);
        }

        if (is_sort(e) || tc.is_prop(e)) {
            return mk_enf_neutral_type();
        }

        if (is_pi(e)) {
            expr it = e;
            while (is_pi(it)) it = binding_body(it);
            if (is_sort(it)) {
                // functions that produce types are irrelevant
                return mk_enf_neutral_type();
            }
        }

        /* If `e` is a trivial structure such as `Subtype`, then convert the
           only relevant field to a runtime type. */
        if (is_app(e)) {
            expr const &fn = get_app_fn(e);
            if (is_constant(fn) && is_inductive(st.env(), const_name(fn))) {
                name const &I_name = const_name(fn);
                environment const &env = st.env();
                if (optional<unsigned> fidx =
                        has_trivial_structure(env, I_name)) {
                    /* Retrieve field `fidx` type */
                    inductive_val I_val = env.get(I_name).to_inductive_val();
                    name K = head(I_val.get_cnstrs());
                    unsigned nparams = I_val.get_nparams();
                    buffer<expr> e_args;
                    get_app_args(e, e_args);
                    lean_assert(nparams <= e_args.size());
```

```
                    expr k_app = mk_app(mk_constant(K, const_levels(fn)),
                                        nparams, e_args.data());
                    expr type = tc.whnf(tc.infer(k_app));
                    local_ctx aux_lctx = lctx;
                    unsigned idx = 0;
                    while (is_pi(type)) {
                        if (idx == *fidx) {
                            return mk_runtime_type(st, aux_lctx,
                                                   binding_domain(type));
                        }
                        expr local = aux_lctx.mk_local_decl(
                            st.ngen(), binding_name(type), binding_domain(type),
                            binding_info(type));
                        type = instantiate(binding_body(type), local);
                        type = type_checker(st, aux_lctx).whnf(type);
                        idx++;
                    }
                }
            }
        }

        return mk_enf_object_type();
    } catch (kernel_exception &) {
        return mk_enf_object_type();
    }
}

environment register_stage1_decl(environment const &env, name const &n,
                                 names const &ls, expr const &t,
                                 expr const &v) {
    declaration aux_decl = mk_definition(mk_cstage1_name(n), ls, t, v,
                                         reducibility_hints::mk_opaque(),
                                         definition_safety::unsafe);
    return env.add(aux_decl, false);
}

bool is_stage2_decl(environment const &env, name const &n) {
    return static_cast<bool>(env.find(mk_cstage2_name(n)));
}

environment register_stage2_decl(environment const &env, name const &n,
                                 expr const &t, expr const &v) {
    declaration aux_decl = mk_definition(mk_cstage2_name(n), names(), t, v,
                                         reducibility_hints::mk_opaque(),
                                         definition_safety::unsafe);
    return env.add(aux_decl, false);
}

/* @[export lean.get_num_lit_core]
   def get_num_lit : expr → option nat */
extern "C" object *lean_get_num_lit(obj_arg o);

optional<nat> get_num_lit_ext(expr const &e) {
    inc(e.raw());
    return to_optional_nat(lean_get_num_lit(e.raw()));
}

optional<unsigned> is_fix_core(name const &n) {
    if (!n.is_atomic() || !n.is_string()) return optional<unsigned>();
    string_ref const &r = n.get_string();
    if (r.length() != 8) return optional<unsigned>();
    char const *s = r.data();
    if (std::strncmp(s, "fixCore", 7) != 0 || !std::isdigit(s[7]))
        return optional<unsigned>();
    return optional<unsigned>(s[7] - '0');
}

optional<expr> mk_enf_fix_core(unsigned n) {
    if (n == 0 || n > 6) return none_expr();
    std::ostringstream s;
```

```
15400     s << "fixCore" << n;
15401     return some_expr(mk_constant(name(s.str())));
15402 }
15403
15404 /* Auxiliary visitor used to detect let-decl LCNF violations.
15405    In LCNF, the type `ty` in `let x : ty := v in t` must not be irrelevant. */
15406 class lcnf_valid_let_decls_fn {
15407     type_checker::state m_st;
15408     local_ctx m_lctx;
15409
15410     environment const &env() const { return m_st.env(); }
15411
15412     name_generator &ngen() { return m_st.ngen(); }
15413
15414     optional<expr> visit_cases(expr const &e) {
15415         buffer<expr> args;
15416         expr const &c = get_app_args(e, args);
15417         unsigned minor_idx;
15418         unsigned minors_end;
15419         bool before_erasure = true;
15420         std::tie(minor_idx, minors_end) =
15421             get_cases_on_minors_range(env(), const_name(c), before_erasure);
15422         for (; minor_idx < minors_end; minor_idx++) {
15423             if (optional<expr> found = visit(args[minor_idx])) return found;
15424         }
15425         return none_expr();
15426     }
15427
15428     optional<expr> visit_app(expr const &e) {
15429         if (is_cases_on_app(env(), e)) {
15430             return visit_cases(e);
15431         } else {
15432             return none_expr();
15433         }
15434     }
15435
15436     optional<expr> visit_lambda(expr e) {
15437         buffer<expr> fvars;
15438         while (is_lambda(e)) {
15439             expr new_d =
15440                 instantiate_rev(binding_domain(e), fvars.size(), fvars.data());
15441             expr new_fvar = m_lctx.mk_local_decl(ngen(), binding_name(e), new_d,
15442                                                  binding_info(e));
15443             fvars.push_back(new_fvar);
15444             e = binding_body(e);
15445         }
15446         e = instantiate_rev(e, fvars.size(), fvars.data());
15447         return visit(e);
15448     }
15449
15450     optional<expr> visit_let(expr e) {
15451         buffer<expr> fvars;
15452         while (is_let(e)) {
15453             expr new_type =
15454                 instantiate_rev(let_type(e), fvars.size(), fvars.data());
15455             if (is_irrelevant_type(m_st, m_lctx, new_type)) {
15456                 return some_expr(e);
15457             }
15458             expr new_val =
15459                 instantiate_rev(let_value(e), fvars.size(), fvars.data());
15460             if (optional<expr> found = visit(new_val)) return found;
15461             expr new_fvar =
15462                 m_lctx.mk_local_decl(ngen(), let_name(e), new_type, new_val);
15463             fvars.push_back(new_fvar);
15464             e = let_body(e);
15465         }
15466         return visit(instantiate_rev(e, fvars.size(), fvars.data()));
15467     }
15468
15469     optional<expr> visit(expr const &e) {
```

```
        switch (e.kind()) {
            case expr_kind::Lambda:
                return visit_lambda(e);
            case expr_kind::Let:
                return visit_let(e);
            case expr_kind::App:
                return visit_app(e);
            default:
                return none_expr();
        }
    }

   public:
    lcnf_valid_let_decls_fn(environment const &env, local_ctx const &lctx)
        : m_st(env), m_lctx(lctx) {}

    optional<expr> operator()(expr const &e) { return visit(e); }
};

optional<expr> lcnf_valid_let_decls(environment const &env, expr const &e) {
    return lcnf_valid_let_decls_fn(env, local_ctx())(e);
}

bool lcnf_check_let_decls(environment const &env, comp_decl const &d) {
    if (optional<expr> v = lcnf_valid_let_decls(env, d.snd())) {
        tout() << "LCNF violation at " << d.fst() << "\n" << *v << "\n";
        return false;
    } else {
        return true;
    }
}

bool lcnf_check_let_decls(environment const &env, comp_decls const &ds) {
    for (comp_decl const &d : ds) {
        if (!lcnf_check_let_decls(env, d)) return false;
    }
    return true;
}

// =======================================
// UInt and USize helper functions

std::vector<pair<name, unsigned>> *g_builtin_scalar_size = nullptr;

expr mk_usize_type() { return *g_usize; }

bool is_usize_type(expr const &e) { return is_constant(e, get_usize_name()); }

optional<unsigned> is_builtin_scalar(expr const &type) {
    if (!is_constant(type)) return optional<unsigned>();
    for (pair<name, unsigned> const &p : *g_builtin_scalar_size) {
        if (const_name(type) == p.first) {
            return optional<unsigned>(p.second);
        }
    }
    return optional<unsigned>();
}

optional<unsigned> is_enum_type(environment const &env, expr const &type) {
    expr const &I = get_app_fn(type);
    if (!is_constant(I)) return optional<unsigned>();
    return is_enum_type(env, const_name(I));
}

// =======================================

expr lcnf_eta_expand(type_checker::state &st, local_ctx lctx, expr e) {
    /* Remark: we do not use `type_checker.eta_expand` because it does not
     * preserve LCNF */
    try {
```

```
15540         buffer<expr> args;
15541         type_checker tc(st, lctx);
15542         expr e_type = tc.whnf(tc.infer(e));
15543         while (is_pi(e_type)) {
15544             expr arg = lctx.mk_local_decl(st.ngen(), binding_name(e_type),
15545                                           binding_domain(e_type),
15546                                           binding_info(e_type));
15547             args.push_back(arg);
15548             e_type = type_checker(st, lctx).whnf(
15549                 instantiate(binding_body(e_type), arg));
15550         }
15551         if (args.empty()) return e;
15552         buffer<expr> fvars;
15553         while (is_let(e)) {
15554             expr type =
15555                 instantiate_rev(let_type(e), fvars.size(), fvars.data());
15556             expr val =
15557                 instantiate_rev(let_value(e), fvars.size(), fvars.data());
15558             expr fvar = lctx.mk_local_decl(st.ngen(), let_name(e), type, val);
15559             fvars.push_back(fvar);
15560             e = let_body(e);
15561         }
15562         e = instantiate_rev(e, fvars.size(), fvars.data());
15563         if (!is_lcnf_atom(e)) {
15564             e = lctx.mk_local_decl(st.ngen(), "_e",
15565                                    type_checker(st, lctx).infer(e), e);
15566             fvars.push_back(e);
15567         }
15568         e = mk_app(e, args);
15569         return lctx.mk_lambda(args, lctx.mk_lambda(fvars, e));
15570     } catch (exception &) {
15571         /* This can happen since previous compilation steps may have
15572            produced type incorrect terms. */
15573         return e;
15574     }
15575 }
15576
15577 void initialize_compiler_util() {
15578     g_neutral_expr = new expr(mk_constant("_neutral"));
15579     mark_persistent(g_neutral_expr->raw());
15580     g_unreachable_expr = new expr(mk_constant("_unreachable"));
15581     mark_persistent(g_unreachable_expr->raw());
15582     g_object_type = new expr(mk_constant("_obj"));
15583     mark_persistent(g_object_type->raw());
15584     g_void_type = new expr(mk_constant("_void"));
15585     mark_persistent(g_void_type->raw());
15586     g_usize = new expr(mk_constant(get_usize_name()));
15587     mark_persistent(g_usize->raw());
15588     g_uint8 = new expr(mk_constant(get_uint8_name()));
15589     mark_persistent(g_uint8->raw());
15590     g_uint16 = new expr(mk_constant(get_uint16_name()));
15591     mark_persistent(g_uint16->raw());
15592     g_uint32 = new expr(mk_constant(get_uint32_name()));
15593     mark_persistent(g_uint32->raw());
15594     g_uint64 = new expr(mk_constant(get_uint64_name()));
15595     mark_persistent(g_uint64->raw());
15596     g_builtin_scalar_size = new std::vector<pair<name, unsigned>>();
15597     g_builtin_scalar_size->emplace_back(get_uint8_name(), 1);
15598     g_builtin_scalar_size->emplace_back(get_uint16_name(), 2);
15599     g_builtin_scalar_size->emplace_back(get_uint32_name(), 4);
15600     g_builtin_scalar_size->emplace_back(get_uint64_name(), 8);
15601     g_builtin_scalar_size->emplace_back(get_float_name(), 8);
15602 }
15603
15604 void finalize_compiler_util() {
15605     delete g_neutral_expr;
15606     delete g_unreachable_expr;
15607     delete g_object_type;
15608     delete g_void_type;
15609     delete g_usize;
```

```
15610     delete g_uint8;
15611     delete g_uint16;
15612     delete g_uint32;
15613     delete g_uint64;
15614     delete g_builtin_scalar_size;
15615 }
15616 }  // namespace lean
15617 // ::::::::::::::
15618 // constructions/brec_on.cpp
15619 // ::::::::::::::
15620 /*
15621 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
15622 Released under Apache 2.0 license as described in the file LICENSE.
15623
15624 Author: Leonardo de Moura
15625 */
15626 #include <lean/sstream.h>
15627
15628 #include "kernel/abstract.h"
15629 #include "kernel/environment.h"
15630 #include "kernel/inductive.h"
15631 #include "kernel/instantiate.h"
15632 #include "kernel/kernel_exception.h"
15633 #include "kernel/type_checker.h"
15634 #include "library/aux_recursors.h"
15635 #include "library/bin_app.h"
15636 #include "library/constructions/util.h"
15637 #include "library/protected.h"
15638 #include "library/reducible.h"
15639 #include "library/suffixes.h"
15640 #include "library/util.h"
15641
15642 namespace lean {
15643 static optional<unsigned> is_typeformer_app(
15644     buffer<name> const &typeformer_names, expr const &e) {
15645     expr const &fn = get_app_fn(e);
15646     if (!is_fvar(fn)) return optional<unsigned>();
15647     unsigned r = 0;
15648     for (name const &n : typeformer_names) {
15649         if (fvar_name(fn) == n) return optional<unsigned>(r);
15650         r++;
15651     }
15652     return optional<unsigned>();
15653 }
15654
15655 static environment mk_below(environment const &env, name const &n,
15656                             bool ibelow) {
15657     if (!is_recursive_datatype(env, n)) return env;
15658     if (is_inductive_predicate(env, n)) return env;
15659     local_ctx lctx;
15660     constant_info ind_info = env.get(n);
15661     inductive_val ind_val = ind_info.to_inductive_val();
15662     name_generator ngen = mk_constructions_name_generator();
15663     unsigned nparams = ind_val.get_nparams();
15664     constant_info rec_info = env.get(mk_rec_name(n));
15665     recursor_val rec_val = rec_info.to_recursor_val();
15666     unsigned nminors = rec_val.get_nminors();
15667     unsigned ntypeformers = rec_val.get_nmotives();
15668     names lps = rec_info.get_lparams();
15669     bool is_reflexive = ind_val.is_reflexive();
15670     level lvl = mk_univ_param(head(lps));
15671     levels lvls = lparams_to_levels(tail(lps));
15672     names blvls;  // universe parameter names of ibelow/below
15673     level rlvl;   // universe level of the resultant type
15674     // The arguments of below (ibelow) are the ones in the recursor - minor
15675     // premises. The universe we map to is also different (l+1 for below of
15676     // reflexive types) and (0 fo ibelow).
15677     expr ref_type;
15678     expr Type_result;
15679     if (ibelow) {
```

```
15680         // we are eliminating to Prop
15681         blvls = tail(lps);
15682         rlvl = mk_level_zero();
15683         ref_type = instantiate_lparam(rec_info.get_type(), param_id(lvl),
15684                                       mk_level_zero());
15685     } else if (is_reflexive) {
15686         blvls = lps;
15687         rlvl = get_datatype_level(ind_info.get_type());
15688         // if rlvl is of the form (max 1 l), then rlvl <- l
15689         if (is_max(rlvl) && is_one(max_lhs(rlvl))) rlvl = max_rhs(rlvl);
15690         rlvl = mk_max(mk_succ(lvl), rlvl);
15691         ref_type = instantiate_lparam(rec_info.get_type(), param_id(lvl),
15692                                       mk_succ(lvl));
15693     } else {
15694         // we can simplify the universe levels for non-reflexive datatypes
15695         blvls = lps;
15696         rlvl = mk_max(mk_level_one(), lvl);
15697         ref_type = rec_info.get_type();
15698     }
15699     Type_result = mk_sort(rlvl);
15700     buffer<expr> ref_args;
15701     to_telescope(lctx, ngen, ref_type, ref_args);
15702     lean_assert(ref_args.size() ==
15703                 nparams + ntypeformers + nminors + ind_val.get_nindices() + 1);
15704
15705     // args contains the below/ibelow arguments
15706     buffer<expr> args;
15707     buffer<name> typeformer_names;
15708     // add parameters and typeformers
15709     for (unsigned i = 0; i < nparams; i++) args.push_back(ref_args[i]);
15710     for (unsigned i = nparams; i < nparams + ntypeformers; i++) {
15711         args.push_back(ref_args[i]);
15712         typeformer_names.push_back(fvar_name(ref_args[i]));
15713     }
15714     // we ignore minor premises in below/ibelow
15715     for (unsigned i = nparams + ntypeformers + nminors; i < ref_args.size();
15716          i++)
15717         args.push_back(ref_args[i]);
15718
15719     // We define below/ibelow using the recursor for this type
15720     levels rec_lvls = cons(mk_succ(rlvl), lvls);
15721     expr rec = mk_constant(rec_info.get_name(), rec_lvls);
15722     for (unsigned i = 0; i < nparams; i++) rec = mk_app(rec, args[i]);
15723     // add type formers
15724     for (unsigned i = nparams; i < nparams + ntypeformers; i++) {
15725         buffer<expr> targs;
15726         to_telescope(lctx, ngen, lctx.get_type(args[i]), targs);
15727         rec = mk_app(rec, lctx.mk_lambda(targs, Type_result));
15728     }
15729     // add minor premises
15730     for (unsigned i = nparams + ntypeformers;
15731          i < nparams + ntypeformers + nminors; i++) {
15732         expr minor = ref_args[i];
15733         expr minor_type = lctx.get_type(minor);
15734         buffer<expr> minor_args;
15735         minor_type = to_telescope(lctx, ngen, minor_type, minor_args);
15736         buffer<expr> prod_pairs;
15737         for (expr &minor_arg : minor_args) {
15738             buffer<expr> minor_arg_args;
15739             expr minor_arg_type = to_telescope(
15740                 env, lctx, ngen, lctx.get_type(minor_arg), minor_arg_args);
15741             if (is_typeformer_app(typeformer_names, minor_arg_type)) {
15742                 expr fst = lctx.get_type(minor_arg);
15743                 minor_arg = lctx.mk_local_decl(
15744                     ngen, lctx.get_local_decl(minor_arg).get_user_name(),
15745                     lctx.mk_pi(minor_arg_args, Type_result));
15746                 expr snd = lctx.mk_pi(minor_arg_args,
15747                                       mk_app(minor_arg, minor_arg_args));
15748                 type_checker tc(env, lctx);
15749                 prod_pairs.push_back(mk_pprod(tc, fst, snd, ibelow));
```

```
15750             }
15751         }
15752         type_checker tc(env, lctx);
15753         expr new_arg =
15754             foldr([&](expr const &a,
15755                       expr const &b) { return mk_pprod(tc, a, b, ibelow); },
15756                   [&]() { return mk_unit(rlvl, ibelow); }, prod_pairs.size(),
15757                   prod_pairs.data());
15758         rec = mk_app(rec, lctx.mk_lambda(minor_args, new_arg));
15759     }
15760
15761     // add indices and major premise
15762     for (unsigned i = nparams + ntypeformers; i < args.size(); i++) {
15763         rec = mk_app(rec, args[i]);
15764     }
15765
15766     name below_name = ibelow ? name{n, "ibelow"} : name{n, "below"};
15767     expr below_type = lctx.mk_pi(args, Type_result);
15768     expr below_value = lctx.mk_lambda(args, rec);
15769
15770     declaration new_d = mk_definition_inferring_unsafe(
15771         env, below_name, blvls, below_type, below_value,
15772         reducibility_hints::mk_abbreviation());
15773     environment new_env = env.add(new_d);
15774     new_env =
15775         set_reducible(new_env, below_name, reducible_status::Reducible, true);
15776     new_env = completion_add_to_black_list(new_env, below_name);
15777     return add_protected(new_env, below_name);
15778 }
15779
15780 environment mk_below(environment const &env, name const &n) {
15781     return mk_below(env, n, false);
15782 }
15783
15784 environment mk_ibelow(environment const &env, name const &n) {
15785     return mk_below(env, n, true);
15786 }
15787
15788 static environment mk_brec_on(environment const &env, name const &n, bool ind) {
15789     if (!is_recursive_datatype(env, n)) return env;
15790     if (is_inductive_predicate(env, n)) return env;
15791     local_ctx lctx;
15792     constant_info ind_info = env.get(n);
15793     inductive_val ind_val = ind_info.to_inductive_val();
15794     name_generator ngen = mk_constructions_name_generator();
15795     unsigned nparams = ind_val.get_nparams();
15796     constant_info rec_info = env.get(mk_rec_name(n));
15797     recursor_val rec_val = rec_info.to_recursor_val();
15798     unsigned nminors = rec_val.get_nminors();
15799     unsigned ntypeformers = rec_val.get_nmotives();
15800     unsigned nmutual = length(ind_val.get_all());
15801     if (ntypeformers != nmutual) {
15802         /* The mutual declaration containing `n` contains nested inductive
15803            datatypes. We don't support this kind of declaration here yet. We
15804            will probably never will :) To support it, we will need to generate
15805            an auxiliary `below` for each nested inductive type since their
15806            default `below` is not good here. For example, at
15807            ```
15808            inductive term
15809            | var : string -> term
15810            | app : string -> list term -> term
15811            ```
15812            The `list.below` is not useful since it will not allow us to recurse
15813            over the nested terms. We need to generate another one using the
15814            auxiliary recursor `term.rec_1` for `list term`.
15815         */
15816         return env;
15817     }
15818     names lps = rec_info.get_lparams();
15819     bool is_reflexive = ind_val.is_reflexive();
```

```
15820     level lvl = mk_univ_param(head(lps));
15821     levels lvls = lparams_to_levels(tail(lps));
15822     level rlvl;
15823     names blps;
15824     levels blvls;  // universe level parameters of brec_on/binduction_on
15825     // The arguments of brec_on (binduction_on) are the ones in the recursor -
15826     // minor premises. The universe we map to is also different (l+1 for below
15827     // of reflexive types) and (0 fo ibelow).
15828     expr ref_type;
15829     if (ind) {
15830         // we are eliminating to Prop
15831         blps = tail(lps);
15832         blvls = lvls;
15833         rlvl = mk_level_zero();
15834         ref_type = instantiate_lparam(rec_info.get_type(), param_id(lvl),
15835                                       mk_level_zero());
15836     } else if (is_reflexive) {
15837         blps = lps;
15838         blvls = cons(lvl, lvls);
15839         rlvl = get_datatype_level(ind_info.get_type());
15840         // if rlvl is of the form (max 1 l), then rlvl <- l
15841         if (is_max(rlvl) && is_one(max_lhs(rlvl))) rlvl = max_rhs(rlvl);
15842         rlvl = mk_max(mk_succ(lvl), rlvl);
15843         // inner_prod, inner_prod_intro, pr1, pr2 do not use the same universe
15844         // levels for reflective datatypes.
15845         ref_type = instantiate_lparam(rec_info.get_type(), param_id(lvl),
15846                                       mk_succ(lvl));
15847     } else {
15848         // we can simplify the universe levels for non-reflexive datatypes
15849         blps = lps;
15850         blvls = cons(lvl, lvls);
15851         rlvl = mk_max(mk_level_one(), lvl);
15852         ref_type = rec_info.get_type();
15853     }
15854     buffer<expr> ref_args;
15855     to_telescope(lctx, ngen, ref_type, ref_args);
15856     lean_assert(ref_args.size() ==
15857                 nparams + ntypeformers + nminors + ind_val.get_nindices() + 1);
15858
15859     // args contains the brec_on/binduction_on arguments
15860     buffer<expr> args;
15861     buffer<name> typeformer_names;
15862     // add parameters and typeformers
15863     for (unsigned i = 0; i < nparams; i++) args.push_back(ref_args[i]);
15864     for (unsigned i = nparams; i < nparams + ntypeformers; i++) {
15865         args.push_back(ref_args[i]);
15866         typeformer_names.push_back(fvar_name(ref_args[i]));
15867     }
15868     // add indices and major premise
15869     for (unsigned i = nparams + ntypeformers + nminors; i < ref_args.size();
15870          i++)
15871         args.push_back(ref_args[i]);
15872     // create below terms (one per datatype)
15873     //    (below.{lvls} params type-formers)
15874     // Remark: it also creates the result type
15875     buffer<expr> belows;
15876     expr result_type;
15877     unsigned k = 0;
15878     for (name const &n1 : ind_val.get_all()) {
15879         if (n1 == n) {
15880             result_type = ref_args[nparams + k];
15881             for (unsigned i = nparams + ntypeformers + nminors;
15882                  i < ref_args.size(); i++)
15883                 result_type = mk_app(result_type, ref_args[i]);
15884         }
15885         k++;
15886         name bname = name(n1, ind ? "ibelow" : "below");
15887         expr below = mk_constant(bname, blvls);
15888         for (unsigned i = 0; i < nparams; i++)
15889             below = mk_app(below, ref_args[i]);
```

```
15890         for (unsigned i = nparams; i < nparams + ntypeformers; i++)
15891             below = mk_app(below, ref_args[i]);
15892         belows.push_back(below);
15893     }
15894     // create functionals (one for each type former)
15895     //     Pi idxs t, below idxs t -> C idxs t
15896     buffer<expr> Fs;
15897     name F_name("F");
15898     for (unsigned i = nparams, j = 0; i < nparams + ntypeformers; i++, j++) {
15899         expr const &C = ref_args[i];
15900         buffer<expr> F_args;
15901         to_telescope(lctx, ngen, lctx.get_type(C), F_args);
15902         expr F_result = mk_app(C, F_args);
15903         expr F_below = mk_app(belows[j], F_args);
15904         F_args.push_back(lctx.mk_local_decl(ngen, "f", F_below));
15905         expr F_type = lctx.mk_pi(F_args, F_result);
15906         expr F = lctx.mk_local_decl(ngen, F_name.append_after(j + 1), F_type);
15907         Fs.push_back(F);
15908         args.push_back(F);
15909     }
15910
15911     // We define brec_on/binduction_on using the recursor for this type
15912     levels rec_lvls = cons(rlvl, lvls);
15913     expr rec = mk_constant(rec_info.get_name(), rec_lvls);
15914     // add parameters to rec
15915     for (unsigned i = 0; i < nparams; i++) rec = mk_app(rec, ref_args[i]);
15916     // add type formers to rec
15917     //     Pi indices t, prod (C ... t) (below ... t)
15918     for (unsigned i = nparams, j = 0; i < nparams + ntypeformers; i++, j++) {
15919         expr const &C = ref_args[i];
15920         buffer<expr> C_args;
15921         to_telescope(lctx, ngen, lctx.get_type(C), C_args);
15922         expr C_t = mk_app(C, C_args);
15923         expr below_t = mk_app(belows[j], C_args);
15924         type_checker tc(env, lctx);
15925         expr prod = mk_pprod(tc, C_t, below_t, ind);
15926         rec = mk_app(rec, lctx.mk_lambda(C_args, prod));
15927     }
15928     // add minor premises to rec
15929     for (unsigned i = nparams + ntypeformers, j = 0;
15930          i < nparams + ntypeformers + nminors; i++, j++) {
15931         expr minor = ref_args[i];
15932         expr minor_type = lctx.get_type(minor);
15933         buffer<expr> minor_args;
15934         minor_type = to_telescope(lctx, ngen, minor_type, minor_args);
15935         buffer<expr> pairs;
15936         for (expr &minor_arg : minor_args) {
15937             buffer<expr> minor_arg_args;
15938             expr minor_arg_type = to_telescope(
15939                 env, lctx, ngen, lctx.get_type(minor_arg), minor_arg_args);
15940             if (auto k = is_typeformer_app(typeformer_names, minor_arg_type)) {
15941                 buffer<expr> C_args;
15942                 get_app_args(minor_arg_type, C_args);
15943                 type_checker tc(env, lctx);
15944                 expr new_minor_arg_type = mk_pprod(
15945                     tc, minor_arg_type, mk_app(belows[*k], C_args), ind);
15946                 minor_arg = lctx.mk_local_decl(
15947                     ngen, lctx.get_local_decl(minor_arg).get_user_name(),
15948                     lctx.mk_pi(minor_arg_args, new_minor_arg_type));
15949                 if (minor_arg_args.empty()) {
15950                     pairs.push_back(minor_arg);
15951                 } else {
15952                     type_checker tc(env, lctx);
15953                     expr r = mk_app(minor_arg, minor_arg_args);
15954                     expr r_1 = lctx.mk_lambda(minor_arg_args,
15955                                               mk_pprod_fst(tc, r, ind));
15956                     expr r_2 = lctx.mk_lambda(minor_arg_args,
15957                                               mk_pprod_snd(tc, r, ind));
15958                     pairs.push_back(mk_pprod_mk(tc, r_1, r_2, ind));
15959                 }
```

```
15960             }
15961         }
15962         type_checker tc(env, lctx);
15963         expr b =
15964             foldr([&](expr const &a,
15965                       expr const &b) { return mk_pprod_mk(tc, a, b, ind); },
15966                   [&]() { return mk_unit_mk(rlvl, ind); }, pairs.size(),
15967                   pairs.data());
15968         unsigned F_idx = *is_typeformer_app(typeformer_names, minor_type);
15969         expr F = Fs[F_idx];
15970         buffer<expr> F_args;
15971         get_app_args(minor_type, F_args);
15972         F_args.push_back(b);
15973         expr new_arg = mk_pprod_mk(tc, mk_app(F, F_args), b, ind);
15974         rec = mk_app(rec, lctx.mk_lambda(minor_args, new_arg));
15975     }
15976     // add indices and major to rec
15977     for (unsigned i = nparams + ntypeformers + nminors; i < ref_args.size();
15978          i++)
15979         rec = mk_app(rec, ref_args[i]);
15980
15981     type_checker tc(env, lctx);
15982     name brec_on_name = name(n, ind ? g_binduction_on : g_brec_on);
15983     expr brec_on_type = lctx.mk_pi(args, result_type);
15984     expr brec_on_value = lctx.mk_lambda(args, mk_pprod_fst(tc, rec, ind));
15985
15986     declaration new_d = mk_definition_inferring_unsafe(
15987         env, brec_on_name, blps, brec_on_type, brec_on_value,
15988         reducibility_hints::mk_abbreviation());
15989     environment new_env = env.add(new_d);
15990     new_env =
15991         set_reducible(new_env, brec_on_name, reducible_status::Reducible, true);
15992     new_env = add_aux_recursor(new_env, brec_on_name);
15993     return add_protected(new_env, brec_on_name);
15994 }
15995
15996 environment mk_brec_on(environment const &env, name const &n) {
15997     return mk_brec_on(env, n, false);
15998 }
15999
16000 environment mk_binduction_on(environment const &env, name const &n) {
16001     return mk_brec_on(env, n, true);
16002 }
16003
16004 extern "C" object *lean_mk_below(object *env, object *n) {
16005     return catch_kernel_exceptions<environment>(
16006         [&]() { return mk_below(environment(env), name(n, true)); });
16007 }
16008
16009 extern "C" object *lean_mk_ibelow(object *env, object *n) {
16010     return catch_kernel_exceptions<environment>(
16011         [&]() { return mk_ibelow(environment(env), name(n, true)); });
16012 }
16013
16014 extern "C" object *lean_mk_brec_on(object *env, object *n) {
16015     return catch_kernel_exceptions<environment>(
16016         [&]() { return mk_brec_on(environment(env), name(n, true)); });
16017 }
16018
16019 extern "C" object *lean_mk_binduction_on(object *env, object *n) {
16020     return catch_kernel_exceptions<environment>(
16021         [&]() { return mk_binduction_on(environment(env), name(n, true)); });
16022 }
16023 }  // namespace lean
16024 // ::::::::::::::
16025 // constructions/cases_on.cpp
16026 // ::::::::::::::
16027 /*
16028 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
16029 Released under Apache 2.0 license as described in the file LICENSE.
```

```
16030
16031 Author: Leonardo de Moura
16032 */
16033 #include <lean/sstream.h>
16034
16035 #include "kernel/abstract.h"
16036 #include "kernel/environment.h"
16037 #include "kernel/inductive.h"
16038 #include "kernel/instantiate.h"
16039 #include "kernel/kernel_exception.h"
16040 #include "library/aux_recursors.h"
16041 #include "library/constants.h"
16042 #include "library/constructions/util.h"
16043 #include "library/protected.h"
16044 #include "library/reducible.h"
16045 #include "library/suffixes.h"
16046
16047 namespace lean {
16048 /** \brief Given a `C := As -> Type`, return `As -> unit` */
16049 static expr mk_pi_unit(expr const &C, expr const &unit) {
16050     if (is_pi(C)) {
16051         return mk_pi(binding_name(C), binding_domain(C),
16052                      mk_pi_unit(binding_body(C), unit));
16053     } else {
16054         return unit;
16055     }
16056 }
16057
16058 /** \brief Given a `C := As -> Type`, return `fun (xs : As), unit` */
16059 static expr mk_fun_unit(expr const &C, expr const &unit) {
16060     if (is_pi(C)) {
16061         return mk_lambda(binding_name(C), binding_domain(C),
16062                          mk_fun_unit(binding_body(C), unit));
16063     } else {
16064         return unit;
16065     }
16066 }
16067
16068 static bool is_type_former_arg(buffer<name> const &C_ids, expr const &arg) {
16069     expr const &fn = get_app_fn(arg);
16070     return is_fvar(fn) &&
16071            std::find(C_ids.begin(), C_ids.end(), fvar_name(fn)) != C_ids.end();
16072 }
16073
16074 environment mk_cases_on(environment const &env, name const &n) {
16075     constant_info ind_info = env.get(n);
16076     if (!ind_info.is_inductive())
16077         throw exception(sstream()
16078                         << "error in '" << g_cases_on << "' generation, '" << n
16079                         << "' is not an inductive datatype");
16080     name cases_on_name(n, g_cases_on);
16081     local_ctx lctx;
16082     inductive_val ind_val = ind_info.to_inductive_val();
16083     name_generator ngen = mk_constructions_name_generator();
16084     name rec_name = mk_rec_name(n);
16085     constant_info rec_info = env.get(rec_name);
16086     lean_assert(rec_info.is_recursor());
16087     recursor_val rec_val = rec_info.to_recursor_val();
16088     unsigned num_indices = rec_val.get_nindices();
16089     unsigned num_minors = rec_val.get_nminors();
16090     unsigned num_motives = rec_val.get_nmotives();
16091     unsigned num_params = rec_val.get_nparams();
16092     buffer<name> ind_names;
16093     to_buffer(rec_val.get_all(), ind_names);
16094     /* Populate `rec_fvars` with a free variable for each recursor argument */
16095     buffer<expr> rec_fvars;
16096     expr rec_type = rec_info.get_type();
16097     while (is_pi(rec_type)) {
16098         expr local = lctx.mk_local_decl(ngen, binding_name(rec_type),
16099                                         binding_domain(rec_type),
```

```
                                        binding_info(rec_type));
        rec_type = instantiate(binding_body(rec_type), local);
        rec_fvars.push_back(local);
    }
    /* Remark `rec_fvars` free variables represent the recursor:
       - Type parameters `As` (size == `num_params`)
       - Motives `Cs`         (size == `num_motives`)
       - Minor premises       (size == `num_minors`)
       - Indices              (size == `num_indices`)
       - Major premise        (size == 1)

       The new `cases_on` recursor will have
       - Type parameters `As` (size == `num_params`)
       - Motive C[i]          (size == 1)
       - Minor premises C[i]  (size == number of constructors of the main type)
       - Indices              (size == `num_indices`)
       - Major premise        (size == 1)
    */

    /* Universe levels */
    levels lvls = lparams_to_levels(rec_info.get_lparams());
    bool elim_to_prop =
        rec_info.get_num_lparams() == ind_info.get_num_lparams();
    level elim_lvl = elim_to_prop ? mk_level_zero() : head(lvls);
    /* We need `unit` when `num_motives` > 0 */
    expr unit = mk_unit(elim_lvl);
    expr star = mk_unit_mk(elim_lvl);

    buffer<expr> cases_on_params;
    expr rec_cnst = mk_constant(rec_name, lvls);
    buffer<expr> rec_args;  // arguments for rec used to define cases_on

    /* Add type parameters `As` */
    for (unsigned i = 0; i < num_params; i++) {
        cases_on_params.push_back(rec_fvars[i]);
        rec_args.push_back(rec_fvars[i]);
    }

    /* Add C[i] */
    buffer<name> C_ids;  // unique ids of all motives (aka type formers)
    unsigned i = num_params;
    name C_main_id;  // unique id of the main type former
    for (unsigned j = 0; j < num_motives; j++) {
        C_ids.push_back(fvar_name(rec_fvars[i]));
        if (j < ind_names.size() && ind_names[j] == n) {
            cases_on_params.push_back(rec_fvars[i]);
            rec_args.push_back(rec_fvars[i]);
            C_main_id = fvar_name(rec_fvars[i]);
        } else {
            rec_args.push_back(mk_fun_unit(lctx.get_type(rec_fvars[i]), unit));
        }
        i++;
    }

    /* Add indices and major-premise */
    for (unsigned i = 0; i < num_indices + 1; i++)
        cases_on_params.push_back(
            rec_fvars[num_params + num_motives + num_minors + i]);

    /* Add minor premises */
    auto process_minor = [&](expr const &minor, bool is_main) {
        buffer<expr> minor_non_rec_params;
        buffer<expr> minor_params;
        local_decl minor_decl = lctx.get_local_decl(minor);
        expr minor_type = minor_decl.get_type();
        while (is_pi(minor_type)) {
            expr curr_type = binding_domain(minor_type);
            expr local =
                lctx.mk_local_decl(ngen, binding_name(minor_type), curr_type,
                                   binding_info(minor_type));
```

```
16170             expr it = curr_type;
16171             while (is_pi(it)) it = binding_body(it);
16172             if (is_type_former_arg(C_ids, it)) {
16173                 if (fvar_name(get_app_fn(it)) == C_main_id) {
16174                     minor_params.push_back(local);
16175                 } else {
16176                     expr new_local = lctx.mk_local_decl(
16177                         ngen, binding_name(minor_type),
16178                         mk_pi_unit(curr_type, unit), binding_info(minor_type));
16179                     minor_params.push_back(new_local);
16180                 }
16181             } else {
16182                 minor_params.push_back(local);
16183                 if (is_main) minor_non_rec_params.push_back(local);
16184             }
16185             minor_type = instantiate(binding_body(minor_type), local);
16186         }
16187         if (is_main) {
16188             expr new_C =
16189                 lctx.mk_local_decl(ngen, minor_decl.get_user_name(),
16190                                    lctx.mk_pi(minor_non_rec_params, minor_type),
16191                                    minor_decl.get_info());
16192             cases_on_params.push_back(new_C);
16193             expr new_C_app = mk_app(new_C, minor_non_rec_params);
16194             expr rec_arg = lctx.mk_lambda(minor_params, new_C_app);
16195             rec_args.push_back(rec_arg);
16196         } else {
16197             rec_args.push_back(lctx.mk_lambda(minor_params, star));
16198         }
16199     };
16200     unsigned minor_idx = 0;
16201     for (name const &J_name : ind_names) {
16202         constant_info J_info = env.get(J_name);
16203         lean_assert(J_info.is_inductive());
16204         inductive_val J_val = J_info.to_inductive_val();
16205         unsigned num_cnstrs = length(J_val.get_cnstrs());
16206         for (unsigned i = 0; i < num_cnstrs; i++) {
16207             expr minor = rec_fvars[num_params + num_motives + minor_idx];
16208             process_minor(minor, J_name == n);
16209             minor_idx++;
16210         }
16211     }
16212     for (; minor_idx < num_minors; minor_idx++) {
16213         expr minor = rec_fvars[num_params + num_motives + minor_idx];
16214         process_minor(minor, false);
16215     }
16216
16217     /* Add indices and major-premise to rec_args */
16218     for (unsigned i = 0; i < num_indices + 1; i++)
16219         rec_args.push_back(
16220             rec_fvars[num_params + num_motives + num_minors + i]);
16221
16222     expr cases_on_type = lctx.mk_pi(cases_on_params, rec_type);
16223     expr cases_on_value =
16224         lctx.mk_lambda(cases_on_params, mk_app(rec_cnst, rec_args));
16225     declaration new_d = mk_definition_inferring_unsafe(
16226         env, cases_on_name, rec_info.get_lparams(), cases_on_type,
16227         cases_on_value, reducibility_hints::mk_abbreviation());
16228     environment new_env = env.add(new_d);
16229     new_env = set_reducible(new_env, cases_on_name, reducible_status::Reducible,
16230                             true);
16231     new_env = add_aux_recursor(new_env, cases_on_name);
16232     return add_protected(new_env, cases_on_name);
16233 }
16234
16235 extern "C" object *lean_mk_cases_on(object *env, object *n) {
16236     return catch_kernel_exceptions<environment>(
16237         [&]() { return mk_cases_on(environment(env), name(n, true)); });
16238 }
16239 }  // namespace lean
```

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

Author: Leonardo de Moura
*/
#include "library/constructions/projection.h"
#include "library/constructions/util.h"

namespace lean {
void initialize_constructions_module() {
    initialize_constructions_util();
    initialize_def_projection();
}

void finalize_constructions_module() {
    finalize_def_projection();
    finalize_constructions_util();
}
}  // namespace lean
// ::::::::::::::
// constructions/no_confusion.cpp
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include "kernel/abstract.h"
#include "kernel/environment.h"
#include "kernel/instantiate.h"
#include "kernel/kernel_exception.h"
#include "kernel/type_checker.h"
#include "library/aux_recursors.h"
#include "library/constants.h"
#include "library/constructions/util.h"
#include "library/protected.h"
#include "library/reducible.h"
#include "library/suffixes.h"
#include "library/util.h"

namespace lean {
static void throw_corrupted(name const &n) {
    throw exception(sstream()
                    << "error in '" << g_no_confusion << "' generation, '" << n
                    << "' inductive datatype declaration is corrupted");
}

static optional<environment> mk_no_confusion_type(environment const &env,
                                                  name const &n) {
    constant_info ind_info = env.get(n);
    if (!ind_info.is_inductive() || !can_elim_to_type(env, n))
        return optional<environment>();
    inductive_val ind_val = ind_info.to_inductive_val();
    local_ctx lctx;
    name_generator ngen = mk_constructions_name_generator();
    unsigned nparams = ind_val.get_nparams();
    constant_info cases_info = env.get(name(n, g_cases_on));
    names lps = cases_info.get_lparams();
    level plvl = mk_univ_param(head(lps));
    levels ilvls = lparams_to_levels(tail(lps));
    level rlvl = plvl;
    expr ind_type = instantiate_type_lparams(ind_info, ilvls);
    /* All inductive datatype parameters and indices are arguments */
```

```
16310     buffer<expr> args;
16311     ind_type = to_telescope(lctx, ngen, ind_type, args,
16312                             some(mk_implicit_binder_info()));
16313     if (!is_sort(ind_type) || args.size() < nparams) throw_corrupted(n);
16314     level ind_lvl = sort_level(ind_type);
16315     unsigned nindices = ind_val.get_nindices();
16316     lean_assert(nindices == args.size() - nparams);
16317     /* Create inductive datatype */
16318     expr I = mk_app(mk_constant(n, ilvls), args);
16319     /* Add (P : Type) */
16320     expr P = lctx.mk_local_decl(ngen, "P", mk_sort(plvl), mk_binder_info());
16321     args.push_back(P);
16322     /* Add v1 and v2 elements of the inductive type */
16323     expr v1 = lctx.mk_local_decl(ngen, "v1", I, mk_binder_info());
16324     expr v2 = lctx.mk_local_decl(ngen, "v2", I, mk_binder_info());
16325     args.push_back(v1);
16326     args.push_back(v2);
16327     expr R = mk_sort(rlvl);
16328     expr Pres = P;
16329     name no_confusion_type_name{n, g_no_confusion_type};
16330     expr no_confusion_type_type = lctx.mk_pi(args, R);
16331     /* Create type former */
16332     buffer<expr> type_former_args;
16333     for (unsigned i = nparams; i < nparams + nindices; i++)
16334         type_former_args.push_back(args[i]);
16335     type_former_args.push_back(v1);
16336     expr type_former = lctx.mk_lambda(type_former_args, R);
16337     /* Create cases_on */
16338     levels clvls = levels(mk_succ(rlvl), ilvls);
16339     expr cases_on = mk_app(
16340         mk_app(mk_constant(cases_info.get_name(), clvls), nparams, args.data()),
16341         type_former);
16342     cases_on = mk_app(cases_on, nindices, args.data() + nparams);
16343     expr cases_on1 = mk_app(cases_on, v1);
16344     expr cases_on2 = mk_app(cases_on, v2);
16345     expr t1 = type_checker(env, lctx).infer(cases_on1);
16346     expr t2 = type_checker(env, lctx).infer(cases_on2);
16347     buffer<expr> outer_cases_on_args;
16348     unsigned idx1 = 0;
16349     while (is_pi(t1)) {
16350         buffer<expr> minor1_args;
16351         expr minor1 =
16352             to_telescope(env, lctx, ngen, binding_domain(t1), minor1_args);
16353         expr curr_t2 = t2;
16354         buffer<expr> inner_cases_on_args;
16355         unsigned idx2 = 0;
16356         while (is_pi(curr_t2)) {
16357             buffer<expr> minor2_args;
16358             expr minor2 = to_telescope(env, lctx, ngen, binding_domain(curr_t2),
16359                                        minor2_args);
16360             if (idx1 != idx2) {
16361                 // infeasible case, constructors do not match
16362                 inner_cases_on_args.push_back(
16363                     lctx.mk_lambda(minor2_args, Pres));
16364             } else {
16365                 if (minor1_args.size() != minor2_args.size())
16366                     throw_corrupted(n);
16367                 buffer<expr> rtype_hyp;
16368                 // add equalities
16369                 for (unsigned i = 0; i < minor1_args.size(); i++) {
16370                     expr lhs = minor1_args[i];
16371                     expr rhs = minor2_args[i];
16372                     expr lhs_type = lctx.get_type(lhs);
16373                     if (!type_checker(env, lctx).is_prop(lhs_type)) {
16374                         expr rhs_type = lctx.get_type(rhs);
16375                         level l = sort_level(
16376                             type_checker(env, lctx).ensure_type(lhs_type));
16377                         expr h_type;
16378                         if (type_checker(env, lctx).is_def_eq(lhs_type,
16379                                                               rhs_type)) {
```

```
16380                             h_type =
16381                                 mk_app(mk_constant(get_eq_name(), levels(l)),
16382                                        lhs_type, lhs, rhs);
16383                         } else {
16384                             h_type =
16385                                 mk_app(mk_constant(get_heq_name(), levels(l)),
16386                                        lhs_type, lhs, rhs_type, rhs);
16387                         }
16388                         name lhs_user_name =
16389                             lctx.get_local_decl(lhs).get_user_name();
16390                         rtype_hyp.push_back(lctx.mk_local_decl(
16391                             ngen, lhs_user_name.append_after("_eq"), h_type,
16392                             mk_binder_info()));
16393                     }
16394                 }
16395                 inner_cases_on_args.push_back(lctx.mk_lambda(
16396                     minor2_args, mk_arrow(lctx.mk_pi(rtype_hyp, P), Pres)));
16397             }
16398             idx2++;
16399             curr_t2 = binding_body(curr_t2);
16400         }
16401         outer_cases_on_args.push_back(lctx.mk_lambda(
16402             minor1_args, mk_app(cases_on2, inner_cases_on_args)));
16403         idx1++;
16404         t1 = binding_body(t1);
16405     }
16406     expr no_confusion_type_value =
16407         lctx.mk_lambda(args, mk_app(cases_on1, outer_cases_on_args));
16408     declaration new_d = mk_definition_inferring_unsafe(
16409         env, no_confusion_type_name, lps, no_confusion_type_type,
16410         no_confusion_type_value, reducibility_hints::mk_abbreviation());
16411     environment new_env = env.add(new_d);
16412     new_env = set_reducible(new_env, no_confusion_type_name,
16413                             reducible_status::Reducible, true);
16414     new_env = completion_add_to_black_list(new_env, no_confusion_type_name);
16415     return some(add_protected(new_env, no_confusion_type_name));
16416 }
16417
16418 environment mk_no_confusion(environment const &env, name const &n) {
16419     optional<environment> env1 = mk_no_confusion_type(env, n);
16420     if (!env1) return env;
16421     environment new_env = *env1;
16422     local_ctx lctx;
16423     name_generator ngen = mk_constructions_name_generator();
16424     constant_info ind_info = new_env.get(n);
16425     inductive_val ind_val = ind_info.to_inductive_val();
16426     unsigned nparams = ind_val.get_nparams();
16427     constant_info no_confusion_type_info =
16428         new_env.get(name{n, g_no_confusion_type});
16429     constant_info cases_info = new_env.get(name(n, g_cases_on));
16430     names lps = no_confusion_type_info.get_lparams();
16431     levels ls = lparams_to_levels(lps);
16432     expr ind_type = instantiate_type_lparams(ind_info, tail(ls));
16433     level ind_lvl = get_datatype_level(ind_type);
16434     expr no_confusion_type_type =
16435         instantiate_type_lparams(no_confusion_type_info, ls);
16436     buffer<expr> args;
16437     expr type = no_confusion_type_type;
16438     type =
16439         to_telescope(lctx, ngen, type, args, some(mk_implicit_binder_info()));
16440     lean_assert(args.size() >= nparams + 3);
16441     unsigned nindices = args.size() - nparams - 3;  // 3 is for P v1 v2
16442     expr range =
16443         mk_app(mk_constant(no_confusion_type_info.get_name(), ls), args);
16444     expr P = args[args.size() - 3];
16445     expr v1 = args[args.size() - 2];
16446     expr v2 = args[args.size() - 1];
16447     expr v_type = lctx.get_type(v1);
16448     level v_lvl = sort_level(type_checker(new_env, lctx).ensure_type(v_type));
16449     expr eq_v = mk_app(mk_constant(get_eq_name(), levels(v_lvl)), v_type);
```

```
16450     expr H12 =
16451         lctx.mk_local_decl(ngen, "h12", mk_app(eq_v, v1, v2), mk_binder_info());
16452     args.push_back(H12);
16453     name no_confusion_name{n, g_no_confusion};
16454     expr no_confusion_ty = lctx.mk_pi(args, range);
16455     // The gen proof is of the form
16456     //   (fun H11 : v1 = v1, cases_on Params (fun Indices v1, no_confusion_type
16457     //   Params Indices P v1 v1) Indices v1
16458     //        <for-each case>
16459     //        (fun H : (equations -> P), H (refl) ... (refl))
16460     //        ...
16461     //   )
16462
16463     // H11 is for creating the generalization
16464     expr H11 =
16465         lctx.mk_local_decl(ngen, "h11", mk_app(eq_v, v1, v1), mk_binder_info());
16466     // Create the type former (fun Indices v1, no_confusion_type Params Indices
16467     // P v1 v1)
16468     buffer<expr> type_former_args;
16469     for (unsigned i = nparams; i < nparams + nindices; i++)
16470         type_former_args.push_back(args[i]);
16471     type_former_args.push_back(v1);
16472     buffer<expr> no_confusion_type_args;
16473     for (unsigned i = 0; i < nparams + nindices; i++)
16474         no_confusion_type_args.push_back(args[i]);
16475     no_confusion_type_args.push_back(P);
16476     no_confusion_type_args.push_back(v1);
16477     no_confusion_type_args.push_back(v1);
16478     expr no_confusion_type_app =
16479         mk_app(mk_constant(no_confusion_type_info.get_name(), ls),
16480                no_confusion_type_args);
16481     expr type_former = lctx.mk_lambda(type_former_args, no_confusion_type_app);
16482     // create cases_on
16483     levels clvls = ls;
16484     expr cases_on = mk_app(
16485         mk_app(mk_constant(cases_info.get_name(), clvls), nparams, args.data()),
16486         type_former);
16487     cases_on = mk_app(mk_app(cases_on, nindices, args.data() + nparams), v1);
16488     expr cot = type_checker(new_env, lctx).infer(cases_on);
16489
16490     while (is_pi(cot)) {
16491         buffer<expr> minor_args;
16492         expr minor =
16493             to_telescope(new_env, lctx, ngen, binding_domain(cot), minor_args);
16494         lean_assert(!minor_args.empty());
16495         expr H = minor_args.back();
16496         expr Ht = lctx.get_type(H);
16497         buffer<expr> refl_args;
16498         while (is_pi(Ht)) {
16499             buffer<expr> eq_args;
16500             expr eq_fn = get_app_args(binding_domain(Ht), eq_args);
16501             if (const_name(eq_fn) == get_eq_name()) {
16502                 refl_args.push_back(
16503                     mk_app(mk_constant(get_eq_refl_name(), const_levels(eq_fn)),
16504                            eq_args[0], eq_args[2]));
16505             } else {
16506                 refl_args.push_back(mk_app(
16507                     mk_constant(get_heq_refl_name(), const_levels(eq_fn)),
16508                     eq_args[0], eq_args[1]));
16509             }
16510             Ht = binding_body(Ht);
16511         }
16512         expr pr = mk_app(H, refl_args);
16513         cases_on = mk_app(cases_on, lctx.mk_lambda(minor_args, pr));
16514         cot = binding_body(cot);
16515     }
16516     expr gen = cases_on;
16517     gen = lctx.mk_lambda(H11, gen);
16518     // Now, we use gen to build the final proof using eq.rec
16519     //
```

```
16520     //  eq.ndrec InductiveType v1 (fun (a : InductiveType), v1 = a ->
16521     //  no_confusion_type Params Indices v1 a) gen v2 H12 H12
16522     //
16523     level eq_rec_l1 = head(ls);
16524     expr eq_rec = mk_app(mk_constant(get_eq_ndrec_name(), {eq_rec_l1, v_lvl}),
16525                          v_type, v1);
16526     // create eq_rec type_former
16527     //    (fun (a : InductiveType), v1 = a -> no_confusion_type Params Indices
16528     //    v1 a)
16529     expr a = lctx.mk_local_decl(ngen, "a", v_type, mk_binder_info());
16530     expr H1a =
16531         lctx.mk_local_decl(ngen, "h1a", mk_app(eq_v, v1, a), mk_binder_info());
16532     // reusing no_confusion_type_args... we just replace the last argument with
16533     // a
16534     no_confusion_type_args.pop_back();
16535     no_confusion_type_args.push_back(a);
16536     expr no_confusion_type_app_1a =
16537         mk_app(mk_constant(no_confusion_type_info.get_name(), ls),
16538                no_confusion_type_args);
16539     expr rec_type_former =
16540         lctx.mk_lambda(a, lctx.mk_pi(H1a, no_confusion_type_app_1a));
16541     // finalize eq_rec
16542     eq_rec = mk_app(mk_app(eq_rec, rec_type_former, gen, v2, H12), H12);
16543     //
16544     expr no_confusion_val = lctx.mk_lambda(args, eq_rec);
16545     declaration new_d = mk_definition_inferring_unsafe(
16546         new_env, no_confusion_name, lps, no_confusion_ty, no_confusion_val,
16547         reducibility_hints::mk_abbreviation());
16548     new_env = new_env.add(new_d);
16549     new_env = set_reducible(new_env, no_confusion_name,
16550                             reducible_status::Reducible, true);
16551     new_env = add_no_confusion(new_env, no_confusion_name);
16552     return add_protected(new_env, no_confusion_name);
16553 }
16554
16555 extern "C" object *lean_mk_no_confusion(object *env, object *n) {
16556     return catch_kernel_exceptions<environment>(
16557         [&]() { return mk_no_confusion(environment(env), name(n, true)); });
16558 }
16559 }  // namespace lean
16560 // ::::::::::::::
16561 // constructions/projection.cpp
16562 // ::::::::::::::
16563 /*
16564 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
16565 Released under Apache 2.0 license as described in the file LICENSE.
16566
16567 Author: Leonardo de Moura
16568 */
16569 #include <lean/sstream.h>
16570
16571 #include <string>
16572
16573 #include "kernel/abstract.h"
16574 #include "kernel/inductive.h"
16575 #include "kernel/instantiate.h"
16576 #include "kernel/kernel_exception.h"
16577 #include "kernel/type_checker.h"
16578 #include "library/class.h"
16579 #include "library/constructions/projection.h"
16580 #include "library/constructions/util.h"
16581 #include "library/projection.h"
16582 #include "library/reducible.h"
16583 #include "library/util.h"
16584
16585 namespace lean {
16586 [[noreturn]] static void throw_ill_formed(name const &n) {
16587     throw exception(sstream() << "projection generation, '" << n
16588                               << "' is an ill-formed inductive datatype");
16589 }
```

```
16590
16591 static bool is_prop(expr type) {
16592     while (is_pi(type)) {
16593         type = binding_body(type);
16594     }
16595     return is_sort(type) && is_zero(sort_level(type));
16596 }
16597
16598 environment mk_projections(environment const &env, name const &n,
16599                            buffer<name> const &proj_names,
16600                            buffer<implicit_infer_kind> const &infer_kinds,
16601                            bool inst_implicit) {
16602     lean_assert(proj_names.size() == infer_kinds.size());
16603     local_ctx lctx;
16604     name_generator ngen = mk_constructions_name_generator();
16605     constant_info ind_info = env.get(n);
16606     if (!ind_info.is_inductive())
16607         throw exception(sstream() << "projection generation, '" << n
16608                                   << "' is not an inductive datatype");
16609     inductive_val ind_val = ind_info.to_inductive_val();
16610     unsigned nparams = ind_val.get_nparams();
16611     name rec_name = mk_rec_name(n);
16612     if (ind_val.get_nindices() > 0)
16613         throw exception(sstream()
16614                         << "projection generation, '" << n
16615                         << "' is an indexed inductive datatype family");
16616     if (length(ind_val.get_cnstrs()) != 1)
16617         throw exception(sstream() << "projection generation, '" << n
16618                                   << "' does not have a single constructor");
16619     constant_info cnstr_info = env.get(head(ind_val.get_cnstrs()));
16620     expr cnstr_type = cnstr_info.get_type();
16621     bool is_predicate = is_prop(ind_info.get_type());
16622     names lvl_params = ind_info.get_lparams();
16623     levels lvls = lparams_to_levels(lvl_params);
16624     buffer<expr> params;  // datatype parameters
16625     for (unsigned i = 0; i < nparams; i++) {
16626         if (!is_pi(cnstr_type)) throw_ill_formed(n);
16627         auto bi = binding_info(cnstr_type);
16628         auto type = binding_domain(cnstr_type);
16629         if (!is_inst_implicit(bi))
16630             // We reset implicit binders in favor of having them inferred by
16631             // `infer_implicit_params` later
16632             bi = mk_binder_info();
16633         if (is_class_out_param(type)) {
16634             // hide `out_param`
16635             type = app_arg(type);
16636             // out_params should always be implicit since they can be inferred
16637             // from the later `c` argument
16638             bi = mk_implicit_binder_info();
16639         }
16640         expr param =
16641             lctx.mk_local_decl(ngen, binding_name(cnstr_type), type, bi);
16642         cnstr_type = instantiate(binding_body(cnstr_type), param);
16643         params.push_back(param);
16644     }
16645     expr C_A = mk_app(mk_constant(n, lvls), params);
16646     binder_info c_bi =
16647         inst_implicit ? mk_inst_implicit_binder_info() : mk_binder_info();
16648     expr c = lctx.mk_local_decl(ngen, name("self"), C_A, c_bi);
16649     buffer<expr> cnstr_type_args;  // arguments that are not parameters
16650     expr it = cnstr_type;
16651     while (is_pi(it)) {
16652         expr local = lctx.mk_local_decl(ngen, binding_name(it),
16653                                         binding_domain(it), binding_info(it));
16654         cnstr_type_args.push_back(local);
16655         it = instantiate(binding_body(it), local);
16656     }
16657     unsigned i = 0;
16658     environment new_env = env;
16659     for (name const &proj_name : proj_names) {
```

```
16660         if (!is_pi(cnstr_type))
16661             throw exception(sstream()
16662                             << "generating projection '" << proj_name << "', '"
16663                             << n << "' does not have sufficient data");
16664         expr result_type = binding_domain(cnstr_type);
16665         if (is_predicate && !type_checker(new_env, lctx).is_prop(result_type)) {
16666             throw exception(sstream() << "failed to generate projection '"
16667                                      << proj_name << "' for '" << n << "', "
16668                                      << "type is an inductive predicate, but "
16669                                         "field is not a proposition");
16670         }
16671         buffer<expr> proj_args;
16672         proj_args.append(params);
16673         proj_args.push_back(c);
16674         expr proj_type = lctx.mk_pi(proj_args, result_type);
16675         proj_type = infer_implicit_params(proj_type, nparams, infer_kinds[i]);
16676         expr proj_val = mk_proj(n, i, c);
16677         proj_val = lctx.mk_lambda(proj_args, proj_val);
16678         declaration new_d = mk_definition_inferring_unsafe(
16679             env, proj_name, lvl_params, proj_type, proj_val,
16680             reducibility_hints::mk_abbreviation());
16681         new_env = new_env.add(new_d);
16682         if (!inst_implicit)
16683             new_env = set_reducible(new_env, proj_name,
16684                                     reducible_status::Reducible, true);
16685         new_env =
16686             save_projection_info(new_env, proj_name, cnstr_info.get_name(),
16687                                  nparams, i, inst_implicit);
16688         expr proj = mk_app(mk_app(mk_constant(proj_name, lvls), params), c);
16689         cnstr_type = instantiate(binding_body(cnstr_type), proj);
16690         i++;
16691     }
16692     return new_env;
16693 }
16694
16695 extern "C" object *lean_mk_projections(object *env, object *struct_name,
16696                                        object *proj_infos,
16697                                        uint8 inst_implicit) {
16698     environment new_env(env);
16699     name n(struct_name);
16700     list_ref<object_ref> ps(proj_infos);
16701     buffer<name> proj_names;
16702     buffer<implicit_infer_kind> infer_kinds;
16703     for (auto p : ps) {
16704         proj_names.push_back(cnstr_get_ref_t<name>(p, 0));
16705         bool infer_mod = cnstr_get_uint8(p.raw(), sizeof(object *));
16706         infer_kinds.push_back(infer_mod ? implicit_infer_kind::Implicit
16707                                         : implicit_infer_kind::RelaxedImplicit);
16708     }
16709     return catch_kernel_exceptions<environment>([&]() {
16710         return mk_projections(new_env, n, proj_names, infer_kinds,
16711                               inst_implicit != 0);
16712     });
16713 }
16714
16715 void initialize_def_projection() {}
16716
16717 void finalize_def_projection() {}
16718 }  // namespace lean
16719 // ::::::::::::::
16720 // constructions/rec_on.cpp
16721 // ::::::::::::::
16722 /*
16723 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
16724 Released under Apache 2.0 license as described in the file LICENSE.
16725
16726 Author: Leonardo de Moura
16727 */
16728 #include <lean/sstream.h>
16729
```

```
16730 #include "kernel/abstract.h"
16731 #include "kernel/environment.h"
16732 #include "kernel/inductive.h"
16733 #include "kernel/instantiate.h"
16734 #include "kernel/kernel_exception.h"
16735 #include "library/aux_recursors.h"
16736 #include "library/constructions/util.h"
16737 #include "library/protected.h"
16738 #include "library/reducible.h"
16739 #include "library/suffixes.h"
16740
16741 namespace lean {
16742 environment mk_rec_on(environment const &env, name const &n) {
16743     constant_info ind_info = env.get(n);
16744     if (!ind_info.is_inductive())
16745         throw exception(sstream()
16746                         << "error in '" << g_rec_on << "' generation, '" << n
16747                         << "' is not an inductive datatype");
16748     name_generator ngen = mk_constructions_name_generator();
16749     local_ctx lctx;
16750     name rec_on_name(n, g_rec_on);
16751     constant_info rec_info = env.get(mk_rec_name(n));
16752     recursor_val rec_val = rec_info.to_recursor_val();
16753     buffer<expr> locals;
16754     expr rec_type = rec_info.get_type();
16755     while (is_pi(rec_type)) {
16756         expr local = lctx.mk_local_decl(ngen, binding_name(rec_type),
16757                                         binding_domain(rec_type),
16758                                         binding_info(rec_type));
16759         rec_type = instantiate(binding_body(rec_type), local);
16760         locals.push_back(local);
16761     }
16762
16763     // locals order
16764     //   As Cs minor_premises indices major-premise
16765
16766     // new_locals order
16767     //   As Cs indices major-premise minor-premises
16768     buffer<expr> new_locals;
16769     unsigned num_indices = rec_val.get_nindices();
16770     unsigned num_minors = rec_val.get_nminors();
16771     unsigned AC_sz = locals.size() - num_minors - num_indices - 1;
16772     for (unsigned i = 0; i < AC_sz; i++) new_locals.push_back(locals[i]);
16773     for (unsigned i = 0; i < num_indices + 1; i++)
16774         new_locals.push_back(locals[AC_sz + num_minors + i]);
16775     for (unsigned i = 0; i < num_minors; i++)
16776         new_locals.push_back(locals[AC_sz + i]);
16777     expr rec_on_type = lctx.mk_pi(new_locals, rec_type);
16778
16779     levels ls = lparams_to_levels(rec_info.get_lparams());
16780     expr rec = mk_constant(rec_info.get_name(), ls);
16781     expr rec_on_val = lctx.mk_lambda(new_locals, mk_app(rec, locals));
16782
16783     environment new_env = env.add(mk_definition_inferring_unsafe(
16784         env, rec_on_name, rec_info.get_lparams(), rec_on_type, rec_on_val,
16785         reducibility_hints::mk_abbreviation()));
16786     new_env =
16787         set_reducible(new_env, rec_on_name, reducible_status::Reducible, true);
16788     new_env = add_aux_recursor(new_env, rec_on_name);
16789     return add_protected(new_env, rec_on_name);
16790 }
16791
16792 extern "C" object *lean_mk_rec_on(object *env, object *n) {
16793     return catch_kernel_exceptions<environment>(
16794         [&]() { return mk_rec_on(environment(env), name(n, true)); });
16795 }
16796 }  // namespace lean
16797 // :::::::::::::
16798 // constructions/util.cpp
16799 // :::::::::::::
```

```
16800 /*
16801 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
16802 Released under Apache 2.0 license as described in the file LICENSE.
16803
16804 Author: Leonardo de Moura
16805 */
16806 #include "kernel/type_checker.h"
16807 #include "library/constants.h"
16808 #include "library/util.h"
16809 #include "util/name_generator.h"
16810
16811 namespace lean {
16812 static name *g_constructions_fresh = nullptr;
16813
16814 extern "C" object *lean_completion_add_to_black_list(object *env, object *n);
16815
16816 environment completion_add_to_black_list(environment const &env,
16817                                           name const &decl_name) {
16818     return environment(lean_completion_add_to_black_list(
16819         env.to_obj_arg(), decl_name.to_obj_arg()));
16820 }
16821
16822 static level get_level(type_checker &ctx, expr const &A) {
16823     expr S = ctx.whnf(ctx.infer(A));
16824     if (!is_sort(S)) throw exception("invalid expression, sort expected");
16825     return sort_level(S);
16826 }
16827
16828 expr mk_and_intro(type_checker &ctx, expr const &Ha, expr const &Hb) {
16829     return mk_app(mk_constant(get_and_intro_name()), ctx.infer(Ha),
16830                   ctx.infer(Hb), Ha, Hb);
16831 }
16832
16833 expr mk_and_left(type_checker &, expr const &H) {
16834     return mk_proj(get_and_name(), nat(0), H);
16835 }
16836
16837 expr mk_and_right(type_checker &, expr const &H) {
16838     return mk_proj(get_and_name(), nat(1), H);
16839 }
16840
16841 expr mk_pprod(type_checker &ctx, expr const &A, expr const &B) {
16842     level l1 = get_level(ctx, A);
16843     level l2 = get_level(ctx, B);
16844     return mk_app(mk_constant(get_pprod_name(), {l1, l2}), A, B);
16845 }
16846
16847 expr mk_pprod_mk(type_checker &ctx, expr const &a, expr const &b) {
16848     expr A = ctx.infer(a);
16849     expr B = ctx.infer(b);
16850     level l1 = get_level(ctx, A);
16851     level l2 = get_level(ctx, B);
16852     return mk_app(mk_constant(get_pprod_mk_name(), {l1, l2}), A, B, a, b);
16853 }
16854
16855 expr mk_pprod_fst(type_checker &, expr const &p) {
16856     return mk_proj(get_pprod_name(), nat(0), p);
16857 }
16858
16859 expr mk_pprod_snd(type_checker &, expr const &p) {
16860     return mk_proj(get_pprod_name(), nat(1), p);
16861 }
16862
16863 expr mk_pprod(type_checker &ctx, expr const &a, expr const &b, bool prop) {
16864     return prop ? mk_and(a, b) : mk_pprod(ctx, a, b);
16865 }
16866
16867 expr mk_pprod_mk(type_checker &ctx, expr const &a, expr const &b, bool prop) {
16868     return prop ? mk_and_intro(ctx, a, b) : mk_pprod_mk(ctx, a, b);
16869 }
```

```
16870
16871 expr mk_pprod_fst(type_checker &ctx, expr const &p, bool prop) {
16872     return prop ? mk_and_left(ctx, p) : mk_pprod_fst(ctx, p);
16873 }
16874
16875 expr mk_pprod_snd(type_checker &ctx, expr const &p, bool prop) {
16876     return prop ? mk_and_right(ctx, p) : mk_pprod_snd(ctx, p);
16877 }
16878
16879 name_generator mk_constructions_name_generator() {
16880     return name_generator(*g_constructions_fresh);
16881 }
16882
16883 void initialize_constructions_util() {
16884     g_constructions_fresh = new name("_cnstr_fresh");
16885     mark_persistent(g_constructions_fresh->raw());
16886     register_name_generator_prefix(*g_constructions_fresh);
16887 }
16888
16889 void finalize_constructions_util() { delete g_constructions_fresh; }
16890 }  // namespace lean
16891 // ::::::::::::::
16892 // annotation.h
16893 // ::::::::::::::
16894 /*
16895 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
16896 Released under Apache 2.0 license as described in the file LICENSE.
16897
16898 Author: Leonardo de Moura
16899 */
16900 #pragma once
16901 #include <string>
16902
16903 #include "kernel/expr.h"
16904
16905 namespace lean {
16906 /** \brief Declare a new kind of annotation. It must only be invoked at startup
16907    time Use helper obect #register_annotation_fn. */
16908 void register_annotation(name const &n);
16909
16910 /** \brief Create an annotated expression with tag \c kind based on \c e.
16911
16912     \pre \c kind must have been registered using #register_annotation.
16913
16914     \remark Annotations have no real semantic meaning, but are useful for
16915    guiding pretty printer and/or automation.
16916 */
16917 expr mk_annotation(name const &kind, expr const &e);
16918 /** \brief Return true iff \c e was created using #mk_annotation. */
16919 optional<expr> is_annotation(expr const &e);
16920 /** \brief Return true iff \c e was created using #mk_annotation, and has tag \c
16921  * kind. */
16922 bool is_annotation(expr const &e, name const &kind);
16923 /** \brief Return true iff \c e is of the form (a_1 ... (a_k e') ...)
16924     where all a_i's are annotations and one of the is \c kind.
16925
16926     \remark is_nested_annotation(e, kind) implies is_annotation(e, kind)
16927 */
16928 bool is_nested_annotation(expr const &e, name const &kind);
16929
16930 /** \brief Return the annotated expression, \c e must have been created using
16931    #mk_annotation.
16932
16933     \post get_annotation_arg(mk_annotation(k, e)) == e
16934 */
16935 expr const &get_annotation_arg(expr const &e);
16936 /** \brief Return the king of the annotated expression, \c e must have been
16937    created using #mk_annotation.
16938
16939     \post get_annotation_arg(mk_annotation(k, e)) == k
```

```
16940 */
16941 name get_annotation_kind(expr const &e);
16942
16943 /** \brief Return the nested annotated expression, \c e must have been created
16944    using #mk_annotation. This function is the "transitive" version of
16945    #get_annotation_arg. It guarantees that the result does not satisfy the
16946    predicate is_annotation.
16947 */
16948 expr const &get_nested_annotation_arg(expr const &e);
16949
16950 /** \brief Copy annotation from \c from to \c to. */
16951 expr copy_annotations(expr const &from, expr const &to);
16952
16953 /** \brief Tag \c e as a 'have'-expression. 'have' is a pre-registered
16954  * annotation. */
16955 expr mk_have_annotation(expr const &e);
16956 /** \brief Tag \c e as a 'show'-expression. 'show' is a pre-registered
16957  * annotation. */
16958 expr mk_show_annotation(expr const &e);
16959 /** \brief Tag \c e as a 'suffices'-expression. 'suffices' is a pre-registered
16960  * annotation. */
16961 expr mk_suffices_annotation(expr const &e);
16962
16963 expr mk_checkpoint_annotation(expr const &e);
16964 /** \brief Return true iff \c e was created using #mk_have_annotation. */
16965 bool is_have_annotation(expr const &e);
16966 /** \brief Return true iff \c e was created using #mk_show_annotation. */
16967 bool is_show_annotation(expr const &e);
16968 /** \brief Return true iff \c e was created using #mk_suffices_annotation. */
16969 bool is_suffices_annotation(expr const &e);
16970 bool is_checkpoint_annotation(expr const &e);
16971
16972 /** \brief Return the serialization 'opcode' for annotation macros. */
16973 std::string const &get_annotation_opcode();
16974
16975 void initialize_annotation();
16976 void finalize_annotation();
16977 }  // namespace lean
16978 // ::::::::::::::
16979 // aux_recursors.h
16980 // ::::::::::::::
16981 /*
16982 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
16983 Released under Apache 2.0 license as described in the file LICENSE.
16984
16985 Author: Leonardo de Moura
16986 */
16987 #pragma once
16988 #include "kernel/environment.h"
16989
16990 namespace lean {
16991 /** \brief Mark \c r as an auxiliary recursor automatically created by the
16992    system. We use it to mark recursors such as brec_on, rec_on, induction_on,
16993    etc. */
16994 environment add_aux_recursor(environment const &env, name const &r);
16995 environment add_no_confusion(environment const &env, name const &r);
16996 /** \brief Return true iff \c n is the name of an auxiliary recursor.
16997     \see add_aux_recursor */
16998 bool is_aux_recursor(environment const &env, name const &n);
16999 bool is_no_confusion(environment const &env, name const &n);
17000 }  // namespace lean
17001 // ::::::::::::::
17002 // bin_app.h
17003 // ::::::::::::::
17004 /*
17005 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17006 Released under Apache 2.0 license as described in the file LICENSE.
17007
17008 Author: Leonardo de Moura
17009 */
```

```
#pragma once
#include "kernel/expr.h"

namespace lean {
/** \brief Return true iff \c t is of the form <tt>((f s1) s2)</tt> */
bool is_bin_app(expr const &t, expr const &f);
/** \brief Return true iff \c t is of the form <tt>((f s1) s2)</tt>, if the
 * result is true, then store a1 -> lhs, a2 -> rhs */
bool is_bin_app(expr const &t, expr const &f, expr &lhs, expr &rhs);

/** \brief Return unit if <tt>num_args == 0</tt>, args[0] if <tt>num_args ==
   1</tt>, and <tt>(op args[0] (op args[1] (op ... )))</tt> */
expr mk_bin_rop(expr const &op, expr const &unit, unsigned num_args,
                expr const *args);
expr mk_bin_rop(expr const &op, expr const &unit,
                std::initializer_list<expr> const &l);

/** \brief Version of foldr that only uses unit when num_args == 0 */
template <typename MkBin, typename MkUnit>
expr foldr_compact(MkBin &&mkb, MkUnit &&mku, unsigned num_args,
                   expr const *args) {
    if (num_args == 0) {
        return mku();
    } else {
        expr r = args[num_args - 1];
        unsigned i = num_args - 1;
        while (i > 0) {
            --i;
            r = mkb(args[i], r);
        }
        return r;
    }
}

/** \brief Version of foldr that only uses unit when num_args == 0 */
template <typename MkBin, typename MkUnit>
expr foldr(MkBin &&mkb, MkUnit &&mku, unsigned num_args, expr const *args) {
    expr r = mku();
    unsigned i = num_args;
    while (i > 0) {
        --i;
        r = mkb(args[i], r);
    }
    return r;
}

/** \brief Return unit if <tt>num_args == 0</tt>, args[0] if <tt>num_args ==
   1</tt>, and <tt>(op ... (op (op args[0] args[1]) args[2]) ...)</tt> */
expr mk_bin_lop(expr const &op, expr const &unit, unsigned num_args,
                expr const *args);
expr mk_bin_lop(expr const &op, expr const &unit,
                std::initializer_list<expr> const &l);
}  // namespace lean
// ::::::::::::::
// class.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "library/util.h"
namespace lean {
/** \brief Return true iff \c c was declared with \c add_class. */
bool is_class(environment const &env, name const &c);
/** \brief Return true iff \c i was declared with \c add_instance. */
bool is_instance(environment const &env, name const &i);

```

```
17080 name const &get_anonymous_instance_prefix();
17081 name mk_anonymous_inst_name(unsigned idx);
17082 bool is_anonymous_inst_name(name const &n);
17083
17084 /** \brief Return true iff e is of the form `outParam a` */
17085 bool is_class_out_param(expr const &e);
17086
17087 /** \brief Return true iff c is a type class that contains an `outParam` */
17088 bool has_class_out_params(environment const &env, name const &c);
17089
17090 void initialize_class();
17091 void finalize_class();
17092 }  // namespace lean
17093 // ::::::::::::::
17094 // constants.h
17095 // ::::::::::::::
17096 // Copyright (c) 2015 Microsoft Corporation. All rights reserved.
17097 // Released under Apache 2.0 license as described in the file LICENSE.
17098 // DO NOT EDIT, automatically generated file, generator
17099 // scripts/gen_constants_cpp.py
17100 #include "util/name.h"
17101 namespace lean {
17102 void initialize_constants();
17103 void finalize_constants();
17104 name const &get_absurd_name();
17105 name const &get_and_name();
17106 name const &get_and_left_name();
17107 name const &get_and_right_name();
17108 name const &get_and_intro_name();
17109 name const &get_and_rec_name();
17110 name const &get_and_cases_on_name();
17111 name const &get_array_name();
17112 name const &get_array_sz_name();
17113 name const &get_array_data_name();
17114 name const &get_auto_param_name();
17115 name const &get_bit0_name();
17116 name const &get_bit1_name();
17117 name const &get_has_of_nat_of_nat_name();
17118 name const &get_byte_array_name();
17119 name const &get_bool_name();
17120 name const &get_bool_false_name();
17121 name const &get_bool_true_name();
17122 name const &get_bool_cases_on_name();
17123 name const &get_cast_name();
17124 name const &get_char_name();
17125 name const &get_congr_arg_name();
17126 name const &get_decidable_name();
17127 name const &get_decidable_is_true_name();
17128 name const &get_decidable_is_false_name();
17129 name const &get_decidable_decide_name();
17130 name const &get_empty_name();
17131 name const &get_empty_rec_name();
17132 name const &get_empty_cases_on_name();
17133 name const &get_exists_name();
17134 name const &get_eq_name();
17135 name const &get_eq_cases_on_name();
17136 name const &get_eq_rec_on_name();
17137 name const &get_eq_rec_name();
17138 name const &get_eq_ndrec_name();
17139 name const &get_eq_refl_name();
17140 name const &get_eq_subst_name();
17141 name const &get_eq_symm_name();
17142 name const &get_eq_trans_name();
17143 name const &get_float_name();
17144 name const &get_float_array_name();
17145 name const &get_false_name();
17146 name const &get_false_rec_name();
17147 name const &get_false_cases_on_name();
17148 name const &get_has_add_add_name();
17149 name const &get_has_neg_neg_name();
```

```
name const &get_has_one_one_name();
name const &get_has_zero_zero_name();
name const &get_heq_name();
name const &get_heq_refl_name();
name const &get_iff_name();
name const &get_iff_refl_name();
name const &get_int_name();
name const &get_int_nat_abs_name();
name const &get_int_dec_lt_name();
name const &get_int_of_nat_name();
name const &get_inline_name();
name const &get_io_name();
name const &get_ite_name();
name const &get_lc_proof_name();
name const &get_lc_unreachable_name();
name const &get_list_name();
name const &get_mut_quot_name();
name const &get_nat_name();
name const &get_nat_succ_name();
name const &get_nat_zero_name();
name const &get_nat_has_zero_name();
name const &get_nat_has_one_name();
name const &get_nat_has_add_name();
name const &get_nat_add_name();
name const &get_nat_dec_eq_name();
name const &get_nat_sub_name();
name const &get_ne_name();
name const &get_not_name();
name const &get_opt_param_name();
name const &get_or_name();
name const &get_panic_name();
name const &get_punit_name();
name const &get_punit_unit_name();
name const &get_pprod_name();
name const &get_pprod_mk_name();
name const &get_pprod_fst_name();
name const &get_pprod_snd_name();
name const &get_propext_name();
name const &get_quot_mk_name();
name const &get_quot_lift_name();
name const &get_sorry_ax_name();
name const &get_string_name();
name const &get_string_data_name();
name const &get_subsingleton_elim_name();
name const &get_task_name();
name const &get_thunk_name();
name const &get_thunk_mk_name();
name const &get_thunk_get_name();
name const &get_true_name();
name const &get_true_intro_name();
name const &get_unit_name();
name const &get_unit_unit_name();
name const &get_uint8_name();
name const &get_uint16_name();
name const &get_uint32_name();
name const &get_uint64_name();
name const &get_usize_name();
}  // namespace lean
// ::::::::::::::
// expr_lt.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

```
17220 #include "util/rb_map.h"
17221
17222 namespace lean {
17223 /** \brief Total order on expressions.
17224
17225     \remark If \c use_hash is true, then we use the hash_code to
17226     partially order expressions. Setting use_hash to false is useful
17227     for testing the code.
17228
17229     \remark If lctx is not nullptr, then we use the local_decl index to compare
17230    local constants.
17231 */
17232 bool is_lt(expr const &a, expr const &b, bool use_hash,
17233            local_ctx const *lctx = nullptr);
17234 /** \brief Similar to is_lt, but universe level parameter names are ignored. */
17235 bool is_lt_no_level_params(expr const &a, expr const &b);
17236 inline bool is_hash_lt(expr const &a, expr const &b) {
17237     return is_lt(a, b, true);
17238 }
17239 inline bool operator<(expr const &a, expr const &b) {
17240     return is_lt(a, b, true);
17241 }
17242 inline bool operator>(expr const &a, expr const &b) {
17243     return is_lt(b, a, true);
17244 }
17245 inline bool operator<=(expr const &a, expr const &b) {
17246     return !is_lt(b, a, true);
17247 }
17248 inline bool operator>=(expr const &a, expr const &b) {
17249     return !is_lt(a, b, true);
17250 }
17251 struct expr_quick_cmp {
17252     typedef expr type;
17253     int operator()(expr const &e1, expr const &e2) const {
17254         return is_lt(e1, e2, true) ? -1 : (e1 == e2 ? 0 : 1);
17255     }
17256 };
17257 struct expr_cmp_no_level_params {
17258     int operator()(expr const &e1, expr const &e2) const;
17259 };
17260
17261 template <typename T>
17262 using rb_expr_map = rb_map<expr, T, expr_quick_cmp>;
17263
17264 typedef rb_tree<expr, expr_quick_cmp> rb_expr_tree;
17265 }  // namespace lean
17266 // ::::::::::::::
17267 // expr_pair.h
17268 // ::::::::::::::
17269 /*
17270 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
17271 Released under Apache 2.0 license as described in the file LICENSE.
17272
17273 Author: Leonardo de Moura
17274 */
17275 #pragma once
17276 #include <lean/hash.h>
17277
17278 #include <utility>
17279
17280 #include "library/expr_lt.h"
17281
17282 namespace lean {
17283 inline bool is_lt(expr_pair const &p1, expr_pair const &p2, bool use_hash) {
17284     return is_lt(p1.first, p2.first, use_hash) ||
17285            (p1.first == p2.first && is_lt(p1.second, p2.second, use_hash));
17286 }
17287 struct expr_pair_quick_cmp {
17288     int operator()(expr_pair const &p1, expr_pair const &p2) const {
17289         return is_lt(p1, p2, true) ? -1 : (p1 == p2 ? 0 : 1);
```

```
17290     }
17291 };
17292 }  // namespace lean
17293 // ::::::::::::::
17294 // expr_pair_maps.h
17295 // ::::::::::::::
17296 /*
17297 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
17298 Released under Apache 2.0 license as described in the file LICENSE.
17299
17300 Author: Leonardo de Moura
17301 */
17302 #pragma once
17303 #include <unordered_map>
17304
17305 #include "kernel/expr.h"
17306 #include "library/expr_pair.h"
17307 namespace lean {
17308 // Map based on structural equality
17309 template <typename T>
17310 using expr_pair_struct_map =
17311     std::unordered_map<expr_pair, T, expr_pair_hash, expr_pair_eq>;
17312 }  // namespace lean
17313 // ::::::::::::::
17314 // expr_unsigned_map.h
17315 // ::::::::::::::
17316 /*
17317 Copyright (c) 2016 Microsoft Corporation. All rights reserved.
17318 Released under Apache 2.0 license as described in the file LICENSE.
17319
17320 Author: Leonardo de Moura
17321 */
17322 #pragma once
17323 #include <unordered_map>
17324
17325 #include "kernel/expr.h"
17326
17327 namespace lean {
17328 /* pair (expression, unsigned int) with cached hash code */
17329 struct expr_unsigned {
17330     expr m_expr;
17331     unsigned m_nargs;
17332     unsigned m_hash;
17333     expr_unsigned(expr const &fn, unsigned nargs)
17334         : m_expr(fn), m_nargs(nargs), m_hash(hash(hash(m_expr), m_nargs)) {}
17335 };
17336
17337 struct expr_unsigned_hash_fn {
17338     unsigned operator()(expr_unsigned const &k) const { return k.m_hash; }
17339 };
17340
17341 struct expr_unsigned_eq_fn {
17342     bool operator()(expr_unsigned const &k1, expr_unsigned const &k2) const {
17343         return k1.m_expr == k2.m_expr && k1.m_nargs == k2.m_nargs;
17344     }
17345 };
17346
17347 /* mapping from (expr, unsigned) -> T */
17348 template <typename T>
17349 using expr_unsigned_map =
17350     typename std::unordered_map<expr_unsigned, T, expr_unsigned_hash_fn,
17351                                 expr_unsigned_eq_fn>;
17352 }  // namespace lean
17353 // ::::::::::::::
17354 // formatter.h
17355 // ::::::::::::::
17356 /*
17357 Copyright (c) 2013 Microsoft Corporation. All rights reserved.
17358 Released under Apache 2.0 license as described in the file LICENSE.
17359
```

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

#include "kernel/expr.h"
#include "util/options.h"

namespace lean {
std::ostream &operator<<(std::ostream &out, expr const &e);
void set_print_fn(std::function<void(std::ostream &, expr const &)> const &fn);

void initialize_formatter();
void finalize_formatter();
}  // namespace lean
// ::::::::::::::
// init_module.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once

namespace lean {
void initialize_library_core_module();
void finalize_library_core_module();
void initialize_library_module();
void finalize_library_module();
}  // namespace lean
// ::::::::::::::
// max_sharing.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include "kernel/expr.h"

namespace lean {
/**
   \brief Functional object for creating expressions with maximally
   shared sub-expressions.
*/
class max_sharing_fn {
    struct imp;
    friend expr max_sharing(expr const &a);
    std::unique_ptr<imp> m_ptr;

   public:
    max_sharing_fn();
    ~max_sharing_fn();

    expr operator()(expr const &a);

    /** \brief Return true iff \c a was already processed by this object. */
    bool already_processed(expr const &a) const;

    /** \brief Clear the cache. */
    void clear();
};

```

```
17430 /**
17431    \brief The resultant expression is structurally identical to the input one,
17432    but it uses maximally shared sub-expressions.
17433 */
17434 expr max_sharing(expr const &a);
17435 }  // namespace lean
17436 // ::::::::::::::
17437 // module.h
17438 // ::::::::::::::
17439 /*
17440 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17441 Released under Apache 2.0 license as described in the file LICENSE.
17442
17443 Authors: Leonardo de Moura, Gabriel Ebner, Sebastian Ullrich
17444 */
17445 #pragma once
17446 #include <lean/optional.h>
17447 #include <lean/serializer.h>
17448
17449 #include <iostream>
17450 #include <string>
17451 #include <utility>
17452 #include <vector>
17453
17454 #include "kernel/environment.h"
17455
17456 namespace lean {
17457 /** \brief Store module using \c env. */
17458 void write_module(environment const &env, std::string const &olean_fn);
17459 }  // namespace lean
17460 // ::::::::::::::
17461 // num.h
17462 // ::::::::::::::
17463 /*
17464 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17465 Released under Apache 2.0 license as described in the file LICENSE.
17466
17467 Author: Leonardo de Moura
17468 */
17469 #include <lean/mpz.h>
17470
17471 #include "kernel/environment.h"
17472
17473 namespace lean {
17474 bool is_const_app(expr const &, name const &, unsigned);
17475
17476 /** \brief Return true iff the given expression encodes a numeral. */
17477 bool is_num(expr const &e);
17478
17479 /** \brief Return true iff is_num(e) or \c e is of the form (- e') where
17480  * to_num(e') */
17481 bool is_signed_num(expr const &e);
17482
17483 bool is_zero(expr const &e);
17484 bool is_one(expr const &e);
17485 optional<expr> is_bit0(expr const &e);
17486 optional<expr> is_bit1(expr const &e);
17487 optional<expr> is_neg(expr const &e);
17488
17489 /** \brief Return true iff \c n is zero, one, bit0 or bit1 */
17490 bool is_numeral_const_name(name const &n);
17491
17492 /** Unfold \c e it is is_zero, is_one, is_bit0 or is_bit1 application */
17493 optional<expr> unfold_num_app(environment const &env, expr const &e);
17494
17495 /** \brief If the given expression encodes a numeral, then convert it back to
17496    mpz numeral. \see from_num */
17497 optional<mpz> to_num(expr const &e);
17498
17499 /** \brief Return true iff n is zero/one/has_zero.zero/has_one.one.
```

```
17500     These constants are used to encode numerals, and some tactics may have to
17501    treat them in a special way.
17502
17503     \remark This kind of hard-coded solution is not ideal. One alternative
17504    solution is to have yet another annotation to let user mark constants that
17505    should be treated in a different way by some tactics. */
17506 bool is_num_leaf_constant(name const &n);
17507
17508 /** \brief Encode \c n as an expression using bit0/bit1/one/zero constants */
17509 expr to_nat_expr(mpz const &n);
17510
17511 void initialize_num();
17512 void finalize_num();
17513 }  // namespace lean
17514 // ::::::::::::::
17515 // print.h
17516 // ::::::::::::::
17517 /*
17518 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17519 Released under Apache 2.0 license as described in the file LICENSE.
17520
17521 Author: Leonardo de Moura
17522 */
17523 #pragma once
17524 #include "kernel/expr.h"
17525 #include "library/formatter.h"
17526
17527 namespace lean {
17528 /** \brief Return true iff \c t contains a constant named \c n or a local
17529  * constant with named (pp or not) \c n */
17530 bool is_used_name(expr const &t, name const &n);
17531 name pick_unused_name(expr const &t, name const &s);
17532 /**
17533     \brief Return the body of the binding \c b, where variable #0 is replaced by
17534    a local constant with a "fresh" name. The name is considered fresh if it is
17535    not used by a constant or local constant occuring in the body of \c b. The
17536    fresh constant is also returned (second return value).
17537
17538     \remark If preserve_type is false, then the local constant will not use
17539    binding_domain.
17540 */
17541 pair<expr, expr> binding_body_fresh(expr const &b, bool preserve_type = false);
17542 pair<expr, expr> let_body_fresh(expr const &b, bool preserve_type = false);
17543
17544 /** \brief Use simple formatter as the default print function */
17545 void init_default_print_fn();
17546
17547 void initialize_print();
17548 void finalize_print();
17549 }  // namespace lean
17550 // ::::::::::::::
17551 // profiling.h
17552 // ::::::::::::::
17553 /*
17554 Copyright (c) 2017 Microsoft Corporation. All rights reserved.
17555 Released under Apache 2.0 license as described in the file LICENSE.
17556
17557 Author: Gabriel Ebner
17558 */
17559 #pragma once
17560 #include <util/options.h>
17561
17562 #include <chrono>
17563
17564 namespace lean {
17565
17566 using second_duration = std::chrono::duration<double>;
17567
17568 bool get_profiler(options const &);
17569 second_duration get_profiling_threshold(options const &);
```

```
17570
17571 void initialize_profiling();
17572 void finalize_profiling();
17573
17574 }  // namespace lean
17575 // ::::::::::::::
17576 // projection.h
17577 // ::::::::::::::
17578 /*
17579 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
17580 Released under Apache 2.0 license as described in the file LICENSE.
17581
17582 Author: Leonardo de Moura
17583 */
17584 #pragma once
17585 #include "kernel/environment.h"
17586
17587 namespace lean {
17588 /** \brief Auxiliary information attached to projections. This information
17589     is used to simplify projection over constructor (efficiently).
17590
17591     That is, given a projection pr_i associated with the constructor mk
17592     where A are parameters, we want to implement the following reduction
17593     efficiently. The idea is to avoid unfolding pr_i.
17594
17595        pr_i A (mk A f_1 ... f_n) ==> f_i
17596
17597     We also use this information in the rewriter/simplifier.
17598 */
17599 class projection_info : public object_ref {
17600    public:
17601     projection_info(name const &c, unsigned nparams, unsigned i,
17602                     bool inst_implicit);
17603     projection_info() : projection_info(name(), 0, 0, false) {}
17604     projection_info(projection_info const &other) : object_ref(other) {}
17605     projection_info(projection_info &&other) : object_ref(other) {}
17606     /* low-level constructors */
17607     explicit projection_info(object *o) : object_ref(o) {}
17608     explicit projection_info(b_obj_arg o, bool b) : object_ref(o, b) {}
17609     explicit projection_info(object_ref const &o) : object_ref(o) {}
17610     projection_info &operator=(projection_info const &other) {
17611         object_ref::operator=(other);
17612         return *this;
17613     }
17614     projection_info &operator=(projection_info &&other) {
17615         object_ref::operator=(other);
17616         return *this;
17617     }
17618     name const &get_constructor() const {
17619         return static_cast<name const &>(cnstr_get_ref(*this, 0));
17620     }
17621     unsigned get_nparams() const {
17622         return static_cast<nat const &>(cnstr_get_ref(*this, 1))
17623             .get_small_value();
17624     }
17625     unsigned get_i() const {
17626         return static_cast<nat const &>(cnstr_get_ref(*this, 2))
17627             .get_small_value();
17628     }
17629     bool is_inst_implicit() const;
17630 };
17631
17632 /** \brief Mark \c p as a projection in the given environment and store that
17633     \c mk is the constructor associated with it, \c nparams is the number of
17634    parameters, and \c i says that \c p is the i-th projection.
17635 */
17636 environment save_projection_info(environment const &env, name const &p,
17637                                  name const &mk, unsigned nparams, unsigned i,
17638                                  bool inst_implicit);
17639
```

```
17640 /** \brief If \c p is a projection in the given environment, then return the
17641    information associated with it (constructor, number of parameters, and
17642    index). If \c p is not a projection, then return nullptr.
17643 */
17644 optional<projection_info> get_projection_info(environment const &env,
17645                                               name const &p);
17646
17647 inline bool is_projection(environment const &env, name const &n) {
17648     return static_cast<bool>(get_projection_info(env, n));
17649 }
17650
17651 /** \brief Return true iff the type named \c S can be viewed as
17652     a structure in the given environment.
17653
17654     If not, generate an error message using \c pos.
17655 */
17656 bool is_structure_like(environment const &env, name const &S);
17657 }  // namespace lean
17658 // ::::::::::::::
17659 // protected.h
17660 // ::::::::::::::
17661 /*
17662 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17663 Released under Apache 2.0 license as described in the file LICENSE.
17664
17665 Author: Leonardo de Moura
17666 */
17667 #pragma once
17668 #include "kernel/environment.h"
17669
17670 namespace lean {
17671 /** \brief Mark \c n as protected, protected declarations are ignored by
17672  * wildcard 'open' command */
17673 environment add_protected(environment const &env, name const &n);
17674 /** \brief Return true iff \c n was marked as protected in the environment \c n.
17675  */
17676 bool is_protected(environment const &env, name const &n);
17677 /** \brief Return the shortest name that can be used to reference the given name
17678  */
17679 name get_protected_shortest_name(name const &n);
17680 }  // namespace lean
17681 // ::::::::::::::
17682 // reducible.h
17683 // ::::::::::::::
17684 /*
17685 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17686 Released under Apache 2.0 license as described in the file LICENSE.
17687
17688 Author: Leonardo de Moura
17689 */
17690 #pragma once
17691 #include <memory>
17692
17693 #include "library/util.h"
17694
17695 namespace lean {
17696 enum class reducible_status { Reducible, Semireducible, Irreducible };
17697 /** \brief Mark the definition named \c n as reducible or not.
17698
17699     The method throws an exception if \c n is
17700       - not a definition
17701       - a theorem
17702       - an opaque definition that was not defined in main module
17703
17704     "Reducible" definitions can be freely unfolded by automation (i.e.,
17705    elaborator, simplifier, etc). We should view it as a hint to automation.
17706 */
17707 environment set_reducible(environment const &env, name const &n,
17708                           reducible_status s, bool persistent);
17709
```

```
reducible_status get_reducible_status(environment const &env, name const &n);

inline bool is_reducible(environment const &env, name const &n) {
    return get_reducible_status(env, n) == reducible_status::Reducible;
}
inline bool is_semireducible(environment const &env, name const &n) {
    return get_reducible_status(env, n) == reducible_status::Semireducible;
}
}  // namespace lean
// ::::::::::::::
// replace_visitor.h
// ::::::::::::::
/*
Copyright (c) 2013 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
/**
   \brief Base class for implementing operations that apply modifications
   on expressions.
   The default behavior is a NOOP, users must create subclasses and
   redefine the visit_* methods. */
class replace_visitor {
   protected:
    typedef expr_bi_map<expr> cache;
    cache m_cache;
    expr save_result(expr const &e, expr &&r, bool shared);
    virtual expr visit_sort(expr const &);
    virtual expr visit_constant(expr const &);
    virtual expr visit_var(expr const &);
    virtual expr visit_meta(expr const &);
    virtual expr visit_fvar(expr const &);
    virtual expr visit_app(expr const &);
    virtual expr visit_binding(expr const &);
    virtual expr visit_lambda(expr const &);
    virtual expr visit_pi(expr const &);
    virtual expr visit_let(expr const &e);
    virtual expr visit_lit(expr const &e);
    virtual expr visit_mdata(expr const &e);
    virtual expr visit_proj(expr const &e);
    virtual expr visit(expr const &);

   public:
    expr operator()(expr const &e) { return visit(e); }
    void clear() { m_cache.clear(); }
};
}  // namespace lean
// ::::::::::::::
// sorry.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
/** \brief Return true iff the given environment contains a sorry macro. */
bool has_sorry(environment const &env);
bool has_sorry(expr const &);
bool has_sorry(declaration const &);
bool has_synthetic_sorry(expr const &);
```

```
17780
17781 /** \brief Return true iff \c e is a sorry macro. */
17782 bool is_sorry(expr const &e);
17783 /** \brief Return true iff \c e is a synthetic sorry macro */
17784 bool is_synthetic_sorry(expr const &e);
17785 /** \brief Type of the sorry macro. */
17786 expr const &sorry_type(expr const &sry);
17787 void initialize_sorry();
17788 void finalize_sorry();
17789 }  // namespace lean
17790 // ::::::::::::::
17791 // suffixes.h
17792 // ::::::::::::::
17793 /*
17794 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
17795 Released under Apache 2.0 license as described in the file LICENSE.
17796
17797 Author: Leonardo de Moura
17798 */
17799 #pragma once
17800
17801 namespace lean {
17802 constexpr char const *g_rec = "rec";
17803 constexpr char const *g_rec_on = "recOn";
17804 constexpr char const *g_brec_on = "brecOn";
17805 constexpr char const *g_binduction_on = "binductionOn";
17806 constexpr char const *g_cases_on = "casesOn";
17807 constexpr char const *g_no_confusion = "noConfusion";
17808 constexpr char const *g_no_confusion_type = "noConfusionType";
17809 }  // namespace lean
17810 // ::::::::::::::
17811 // time_task.h
17812 // ::::::::::::::
17813 /*
17814 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
17815 Released under Apache 2.0 license as described in the file LICENSE.
17816
17817 Author: Sebastian Ullrich
17818 */
17819 #pragma once
17820 #include <string>
17821
17822 #include "library/profiling.h"
17823 #include "util/message_definitions.h"
17824 #include "util/timeit.h"
17825
17826 namespace lean {
17827 void report_profiling_time(std::string const &category, second_duration time);
17828 void display_cumulative_profiling_times(std::ostream &out);
17829
17830 /** Measure time of some task and report it for the final cumulative profile. */
17831 class time_task {
17832     std::string m_category;
17833     optional<xtimeit> m_timeit;
17834     time_task *m_parent_task;
17835
17836    public:
17837     time_task(std::string const &category, options const &opts,
17838               name decl = name());
17839     ~time_task();
17840 };
17841
17842 void initialize_time_task();
17843 void finalize_time_task();
17844 }  // namespace lean
17845 // ::::::::::::::
17846 // trace.h
17847 // ::::::::::::::
17848 /*
17849 Copyright (c) 2015 Microsoft Corporation. All rights reserved.
```

```
17850 Released under Apache 2.0 license as described in the file LICENSE.
17851
17852 Author: Leonardo de Moura
17853 */
17854 #pragma once
17855 #include <memory>
17856 #include <string>
17857
17858 #include "kernel/environment.h"
17859 #include "util/message_definitions.h"
17860 #include "util/options.h"
17861
17862 namespace lean {
17863 void register_trace_class(name const &n);
17864 void register_trace_class_alias(name const &n, name const &alias);
17865 bool is_trace_enabled();
17866 bool is_trace_class_enabled(name const &n);
17867
17868 #define lean_is_trace_enabled(CName) \
17869     (::lean::is_trace_enabled() && ::lean::is_trace_class_enabled(CName))
17870
17871 class scope_trace_env {
17872     unsigned m_enable_sz;
17873     unsigned m_disable_sz;
17874     environment const *m_old_env;
17875     options const *m_old_opts;
17876     void init(environment *env, options *opts);
17877
17878    public:
17879     scope_trace_env(environment const &env, options const &opts);
17880     ~scope_trace_env();
17881 };
17882
17883 struct tclass {
17884     name m_cls;
17885     tclass(name const &c) : m_cls(c) {}
17886 };
17887
17888 std::ostream &tout();
17889 std::ostream &operator<<(std::ostream &ios, tclass const &);
17890
17891 #define lean_trace(CName, CODE)             \
17892     {                                       \
17893         if (lean_is_trace_enabled(CName)) { \
17894             tout() << tclass(CName);        \
17895             CODE                            \
17896         }                                   \
17897     }
17898
17899 void trace_expr(environment const &env, options const &opts, expr const &e);
17900 std::string trace_pp_expr(expr const &e);
17901
17902 void initialize_trace();
17903 void finalize_trace();
17904 }  // namespace lean
17905 // ::::::::::::::
17906 // util.h
17907 // ::::::::::::::
17908 /*
17909 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
17910 Released under Apache 2.0 license as described in the file LICENSE.
17911
17912 Author: Leonardo de Moura
17913 */
17914 #pragma once
17915 #include <string>
17916
17917 #include "kernel/environment.h"
17918 #include "library/expr_pair.h"
17919
```

```
17920 namespace lean {
17921 /* If \c n is not in \c env, then return \c. Otherwise, find the first j >= idx
17922    s.t. n.append_after(j) is not in \c env. */
17923 name mk_unused_name(environment const &env, name const &n, unsigned &idx);
17924
17925 /* If \c n is not in \c env, then return \c. Otherwise, find the first j >= 1
17926    s.t. n.append_after(j) is not in \c env. */
17927 name mk_unused_name(environment const &env, name const &n);
17928
17929 /** \brief Return the "arity" of the given type. The arity is the number of
17930  * nested pi-expressions. */
17931 unsigned get_arity(expr type);
17932
17933 optional<expr> is_optional_param(expr const &e);
17934
17935 optional<expr_pair> is_auto_param(expr const &e);
17936
17937 /** brief Return a name that does not appear in `lp_names`. */
17938 name mk_fresh_lp_name(names const &lp_names);
17939
17940 /** \brief Return true iff \c n occurs in \c m */
17941 bool occurs(expr const &n, expr const &m);
17942 /** \brief Return true iff there is a constant named \c n in \c m. */
17943 bool occurs(name const &n, expr const &m);
17944
17945 /** \brief Return true iff t is a constant named f_name or an application of the
17946  * form (f_name a_1 ... a_k) */
17947 bool is_app_of(expr const &t, name const &f_name);
17948 /** \brief Return true iff t is a constant named f_name or an application of the
17949  * form (f_name a_1 ... a_nargs) */
17950 bool is_app_of(expr const &t, name const &f_name, unsigned nargs);
17951
17952 /** \brief If type is of the form (auto_param A p) or (opt_param A v), return A.
17953  * Otherwise, return type. */
17954 expr consume_auto_opt_param(expr const &type);
17955
17956 /** \brief Unfold constant \c e or constant application (i.e., \c e is of the
17957    form (f ....), where \c f is a constant */
17958 optional<expr> unfold_term(environment const &env, expr const &e);
17959 /** \brief If \c e is of the form <tt>(f a_1 ... a_n)</tt>, where \c f is
17960     a non-opaque definition, then unfold \c f, and beta reduce.
17961     Otherwise, return none. */
17962 optional<expr> unfold_app(environment const &env, expr const &e);
17963
17964 /** \brief Reduce (if possible) universe level by 1.
17965     \pre is_not_zero(l) */
17966 optional<level> dec_level(level const &l);
17967
17968 bool has_punit_decls(environment const &env);
17969 bool has_pprod_decls(environment const &env);
17970 bool has_eq_decls(environment const &env);
17971 bool has_heq_decls(environment const &env);
17972 bool has_and_decls(environment const &env);
17973
17974 inline bool is_inductive(environment const &env, name const &n) {
17975     if (optional<constant_info> info = env.find(n)) return info->is_inductive();
17976     return false;
17977 }
17978
17979 inline bool is_constructor(environment const &env, name const &n) {
17980     if (optional<constant_info> info = env.find(n))
17981         return info->is_constructor();
17982     return false;
17983 }
17984
17985 inline bool is_recursor(environment const &env, name const &n) {
17986     if (optional<constant_info> info = env.find(n)) return info->is_recursor();
17987     return false;
17988 }
17989
```

```
17990 /** \brief Return true iff \c n is the name of a recursive datatype in \c env.
17991     That is, it must be an inductive datatype AND contain a recursive
17992    constructor.
17993
17994     \remark Records are inductive datatypes, but they are not recursive.
17995
17996     \remark For mutually indutive datatypes, \c n is considered recursive
17997     if there is a constructor taking \c n. */
17998 bool is_recursive_datatype(environment const &env, name const &n);
17999
18000 /** \brief Return true iff \c n is an inductive predicate, i.e., an inductive
18001    datatype that is in Prop.
18002
18003     \remark If \c env does not have Prop (i.e., Type.{0} is not impredicative),
18004    then this method always return false. */
18005 bool is_inductive_predicate(environment const &env, name const &n);
18006
18007 /** \brief Return true iff \c n is an inductive type with a recursor with an
18008  * extra level parameter. */
18009 bool can_elim_to_type(environment const &env, name const &n);
18010
18011 /** \brief Store in `result` the constructors of the given inductive datatype.
18012     \remark this procedure does nothing if `n` is not an inductive datatype. */
18013 void get_constructor_names(environment const &env, name const &n,
18014                            buffer<name> &result);
18015
18016 /** \brief If \c e is a constructor application, then return the name of the
18017    constructor. Otherwise, return none. */
18018 optional<name> is_constructor_app(environment const &env, expr const &e);
18019
18020 /** \brief If \c e is a constructor application, or a definition that wraps a
18021     constructor application, then return the name of the constructor.
18022     Otherwise, return none. */
18023 optional<name> is_constructor_app_ext(environment const &env, expr const &e);
18024
18025 /** \brief Store in \c result a bit-vector indicating which fields of the
18026    constructor \c n are (computationally) relevant. \pre
18027    inductive::is_intro_rule(env, n) */
18028 void get_constructor_relevant_fields(environment const &env, name const &n,
18029                                      buffer<bool> &result);
18030
18031 /** Return the number of constructors of the given inductive datatype */
18032 unsigned get_num_constructors(environment const &env, name const &n);
18033
18034 /** \brief Return the index (position) of the given constructor in the inductive
18035    datatype declaration. \pre inductive::is_intro_rule(env, n) */
18036 unsigned get_constructor_idx(environment const &env, name const &n);
18037
18038 /** \brief Given a constructor name, return the associated inductive type name.
18039
18040     \pre inductive::is_intro_rule(env, ctor_name) */
18041 name get_constructor_inductive_type(environment const &env,
18042                                     name const &ctor_name);
18043
18044 /** \brief Return the universe where inductive datatype resides
18045     \pre \c ind_type is of the form <tt>Pi (a_1 : A_1) (a_2 : A_2[a_1]) ...,
18046    Type.{lvl}</tt> */
18047 level get_datatype_level(expr const &ind_type);
18048
18049 /** \brief "Consume" Pi-type `type`. This procedure creates free variables based
18050    on the domain of `type` using `lctx`, and store them in telescope and updates
18051    . If `binfo` is provided, then the free variables are annotated with the
18052    given `binder_info`, otherwise the procedure uses the one attached in the
18053    domain. The procedure returns the "body" of type. */
18054 expr to_telescope(local_ctx &lctx, name_generator &ngen, expr const &type,
18055                   buffer<expr> &telescope,
18056                   optional<binder_info> const &binfo = optional<binder_info>());
18057
18058 /** \brief Similar to previous procedure, but uses whnf to check whether `type`
18059  * reduces to `Pi` or not. */
```

```
18060 expr to_telescope(environment const &env, local_ctx &lctx, name_generator &ngen,
18061                     expr type, buffer<expr> &telescope,
18062                     optional<binder_info> const &binfo = optional<binder_info>());
18063
18064 /** \brief Update the result sort of the given type */
18065 expr update_result_sort(expr t, level const &l);
18066
18067 expr instantiate_lparam(expr const &e, name const &p, level const &l);
18068
18069 expr mk_true();
18070 bool is_true(expr const &e);
18071 expr mk_true_intro();
18072
18073 bool is_and(expr const &e, expr &arg1, expr &arg2);
18074 bool is_and(expr const &e);
18075 expr mk_and(expr const &a, expr const &b);
18076
18077 expr mk_unit(level const &l);
18078 expr mk_unit_mk(level const &l);
18079 expr mk_unit();
18080 expr mk_unit_mk();
18081
18082 expr mk_unit(level const &l, bool prop);
18083 expr mk_unit_mk(level const &l, bool prop);
18084
18085 expr mk_nat_type();
18086 bool is_nat_type(expr const &e);
18087 expr mk_nat_zero();
18088 expr mk_nat_one();
18089 expr mk_nat_bit0(expr const &e);
18090 expr mk_nat_bit1(expr const &e);
18091 expr mk_nat_add(expr const &e1, expr const &e2);
18092
18093 expr mk_int_type();
18094 bool is_int_type(expr const &e);
18095
18096 expr mk_char_type();
18097
18098 bool is_ite(expr const &e, expr &c, expr &H, expr &A, expr &t, expr &f);
18099 bool is_ite(expr const &e);
18100
18101 bool is_iff(expr const &e);
18102 bool is_iff(expr const &e, expr &lhs, expr &rhs);
18103 expr mk_iff(expr const &lhs, expr const &rhs);
18104 expr mk_iff_refl(expr const &a);
18105
18106 expr mk_propext(expr const &lhs, expr const &rhs, expr const &iff_pr);
18107
18108 /** \brief Return true iff \c e is a term of the form (eq.rec ....) */
18109 bool is_eq_rec_core(expr const &e);
18110 /** \brief Return true iff \c e is a term of the form (eq.rec ....) */
18111 bool is_eq_rec(expr const &e);
18112 /** \brief Return true iff \c e is a term of the form (eq.drec ....) */
18113 bool is_eq_drec(expr const &e);
18114
18115 bool is_eq(expr const &e);
18116 bool is_eq(expr const &e, expr &lhs, expr &rhs);
18117 bool is_eq(expr const &e, expr &A, expr &lhs, expr &rhs);
18118 /** \brief Return true iff \c e is of the form (eq A a a) */
18119 bool is_eq_a_a(expr const &e);
18120
18121 bool is_heq(expr const &e);
18122 bool is_heq(expr const &e, expr &lhs, expr &rhs);
18123 bool is_heq(expr const &e, expr &A, expr &lhs, expr &B, expr &rhs);
18124
18125 expr mk_false();
18126 expr mk_empty();
18127
18128 bool is_false(expr const &e);
18129 bool is_empty(expr const &e);
```

```
18130
18131 bool is_or(expr const &e);
18132 bool is_or(expr const &e, expr &A, expr &B);
18133
18134 /** \brief Return true if \c e is of the form <tt>(not arg)</tt> or <tt>arg ->
18135    false</tt>, and store \c arg in \c a. Return false otherwise */
18136 bool is_not(expr const &e, expr &a);
18137 inline bool is_not(expr const &e) {
18138     expr a;
18139     return is_not(e, a);
18140 }
18141 /** \brief Extends is_not to handle (lhs ≠ rhs). In the new case, it stores (lhs
18142  * = rhs) in \c a. */
18143 bool is_not_or_ne(expr const &e, expr &a);
18144 expr mk_not(expr const &e);
18145
18146 /** \brief Returns none if \c e is not an application with at least two
18147    arguments. Otherwise it returns <tt>app_fn(app_fn(e))</tt>. */
18148 optional<expr> get_binary_op(expr const &e);
18149 optional<expr> get_binary_op(expr const &e, expr &arg1, expr &arg2);
18150
18151 /** \brief Makes n-ary (right-associative) application. */
18152 expr mk_nary_app(expr const &op, buffer<expr> const &nary_args);
18153 expr mk_nary_app(expr const &op, unsigned num_nary_args, expr const *nary_args);
18154
18155 expr mk_bool();
18156 expr mk_bool_true();
18157 expr mk_bool_false();
18158 expr to_bool_expr(bool b);
18159
18160 /* Similar to is_head_beta, but ignores annotations around the function. */
18161 bool is_annotated_head_beta(expr const &t);
18162 /* Similar to head_beta_reduce, but also reduces annotations around the
18163  * function. */
18164 expr annotated_head_beta_reduce(expr const &t);
18165
18166 bool is_exists(expr const &e, expr &A, expr &p);
18167 bool is_exists(expr const &e);
18168
18169 expr try_eta(expr const &e);
18170 expr beta_reduce(expr t);
18171 expr eta_reduce(expr t);
18172 expr beta_eta_reduce(expr t);
18173
18174 enum class implicit_infer_kind { Implicit, RelaxedImplicit };
18175
18176 /** \brief Infer implicit parameter annotations for the first \c nparams using
18177    mode specified by \c k. */
18178 expr infer_implicit_params(expr const &type, unsigned nparams,
18179                            implicit_infer_kind k);
18180
18181 /** Given an inductive datatype named \c n, return a recursor for \c n that
18182    supports dependent elimination even if \c n is an inductive predicate. */
18183 name get_dep_recursor(environment const &env, name const &n);
18184
18185 /** Given an inductive datatype named \c n, return a cases_on recursor \c n that
18186    supports dependent elimination even if \c n is an inductive predicate. */
18187 name get_dep_cases_on(environment const &env, name const &n);
18188
18189 /** We generate auxiliary unsafe definitions for regular recursive definitions.
18190     The auxiliary unsafe definition has a clear runtime cost execution model,
18191    and we use it in the VM and code generators. This function returns an
18192    auxiliary unsafe definition for the given name. */
18193 name mk_unsafe_rec_name(name const &n);
18194
18195 /** Return some(n') if \c n is a name created using mk_unsafe_rec_name(n') */
18196 optional<name> is_unsafe_rec_name(name const &n);
18197
18198 std::string const &get_version_string();
18199
```

```
expr const &extract_mdata(expr const &);

optional<expr> to_optional_expr(obj_arg o);

void initialize_library_util();
void finalize_library_util();
}  // namespace lean
// ::::::::::::::
// compiler/borrowed_annotation.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "library/compiler/util.h"
namespace lean {
expr mk_borrowed(expr const &e);
bool is_borrowed(expr const &e);
expr get_borrowed_arg(expr const &e);
void initialize_borrowed_annotation();
void finalize_borrowed_annotation();
}  // namespace lean
// ::::::::::::::
// compiler/closed_term_cache.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
namespace lean {
optional<name> get_closed_term_name(environment const &env, expr const &e);
environment cache_closed_term_name(environment const &env, expr const &e,
                                   name const &n);
}  // namespace lean
// ::::::::::::::
// compiler/compiler.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
namespace lean {
environment compile(environment const &env, options const &opts, names cs);
inline environment compile(environment const &env, options const &opts,
                           name const &c) {
    return compile(env, opts, names(c));
}
void initialize_compiler();
void finalize_compiler();
}  // namespace lean
// ::::::::::::::
// compiler/cse.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
```

```
#pragma once
#include "kernel/environment.h"
namespace lean {
/* Common subexpression elimination */
expr cse_core(environment const &env, expr const &e, bool before_erasure);
inline expr cse(environment const &env, expr const &e) {
    return cse_core(env, e, true);
}
inline expr ecse(environment const &env, expr const &e) {
    return cse_core(env, e, false);
}
/* Common case elimination */
expr cce_core(environment const &env, local_ctx const &lctx, expr const &e);
inline expr cce(environment const &env, expr const &e) {
    return cce_core(env, local_ctx(), e);
}
void initialize_cse();
void finalize_cse();
}  // namespace lean
// ::::::::::::::
// compiler/csimp.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
namespace lean {
struct csimp_cfg {
    /* If `m_inline` == false, then we will not inline `c` even if it is marked
     * with the attribute `[inline]`. */
    bool m_inline;
    /* We inline "cheap" functions. We say a function is cheap if
       `get_lcnf_size(val) < m_inline_threshold`, and it is not marked as
       `[noinline]`. */
    unsigned m_inline_threshold;
    /* We only perform float cases_on from cases_on and other expression if the
       potential code blowup is smaller than m_float_cases_threshold. */
    unsigned m_float_cases_threshold;
    /* We inline join-points that are smaller m_inline_threshold. */
    unsigned m_inline_jp_threshold;

   public:
    csimp_cfg(options const &opts);
    csimp_cfg();
};

expr csimp_core(environment const &env, local_ctx const &lctx, expr const &e,
                bool before_erasure, csimp_cfg const &cfg);
inline expr csimp(environment const &env, expr const &e,
                  csimp_cfg const &cfg = csimp_cfg()) {
    return csimp_core(env, local_ctx(), e, true, cfg);
}
inline expr cesimp(environment const &env, expr const &e,
                   csimp_cfg const &cfg = csimp_cfg()) {
    return csimp_core(env, local_ctx(), e, false, cfg);
}
}  // namespace lean
// ::::::::::::::
// compiler/eager_lambda_lifting.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
```

```
#pragma once
#include "kernel/environment.h"
#include "library/compiler/csimp.h"
namespace lean {
/** \brief Eager version of lambda lifting. See comment at
 * eager_lambda_lifting.cpp. */
pair<environment, comp_decls> eager_lambda_lifting(environment env,
                                                   comp_decls const &ds,
                                                   csimp_cfg const &cfg);
/* Return true iff `fn` is the name of an auxiliary function generated by
 * `eager_lambda_lifting`. */
bool is_elambda_lifting_name(name fn);
};  // namespace lean
// ::::::::::::::
// compiler/elim_dead_let.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/expr.h"
namespace lean {
expr elim_dead_let(expr const &e);
void initialize_elim_dead_let();
void finalize_elim_dead_let();
}  // namespace lean
// ::::::::::::::
// compiler/erase_irrelevant.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
namespace lean {
expr erase_irrelevant_core(environment const &env, local_ctx const &lctx,
                           expr const &e);
inline expr erase_irrelevant(environment const &env, expr const &e) {
    return erase_irrelevant_core(env, local_ctx(), e);
}
}  // namespace lean
// ::::::::::::::
// compiler/export_attribute.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
namespace lean {
optional<name> get_export_name_for(environment const &env, name const &n);
inline bool has_export_name(environment const &env, name const &n) {
    return static_cast<bool>(get_export_name_for(env, n));
}
}  // namespace lean
// ::::::::::::::
// compiler/extern_attribute.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
```

```
18410
18411 Authors: Leonardo de Moura
18412 */
18413 #pragma once
18414 #include <string>
18415
18416 #include "kernel/environment.h"
18417 namespace lean {
18418 bool is_extern_constant(environment const &env, name const &c);
18419 optional<expr> get_extern_constant_ll_type(environment const &env,
18420                                            name const &c);
18421 optional<unsigned> get_extern_constant_arity(environment const &env,
18422                                              name const &c);
18423 typedef object_ref extern_attr_data_value;
18424 optional<extern_attr_data_value> get_extern_attr_data(environment const &env,
18425                                                       name const &c);
18426 /* Return true if `c` is an extern constant, and store in borrowed_args and
18427    borrowed_res which arguments/results are marked as borrowed. */
18428 bool get_extern_borrowed_info(environment const &env, name const &c,
18429                               buffer<bool> &borrowed_args, bool &borrowed_res);
18430 }  // namespace lean
18431 // ::::::::::::::
18432 // compiler/extract_closed.h
18433 // ::::::::::::::
18434 /*
18435 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18436 Released under Apache 2.0 license as described in the file LICENSE.
18437
18438 Author: Leonardo de Moura
18439 */
18440 #pragma once
18441 #include "kernel/environment.h"
18442 #include "library/compiler/util.h"
18443 namespace lean {
18444 bool is_extract_closed_aux_fn(name const &n);
18445 pair<environment, comp_decls> extract_closed(environment env,
18446                                              comp_decls const &ds);
18447 }  // namespace lean
18448 // ::::::::::::::
18449 // compiler/find_jp.h
18450 // ::::::::::::::
18451 /*
18452 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
18453 Released under Apache 2.0 license as described in the file LICENSE.
18454
18455 Author: Leonardo de Moura
18456 */
18457 #pragma once
18458 #include "kernel/environment.h"
18459 namespace lean {
18460 expr find_jp(environment const &env, expr const &e);
18461 }
18462 // ::::::::::::::
18463 // compiler/implemented_by_attribute.h
18464 // ::::::::::::::
18465 /*
18466 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
18467 Released under Apache 2.0 license as described in the file LICENSE.
18468
18469 Author: Leonardo de Moura
18470 */
18471 #pragma once
18472 #include "kernel/environment.h"
18473
18474 namespace lean {
18475 optional<name> get_implemented_by_attribute(environment const &env,
18476                                              name const &n);
18477 }
18478 // ::::::::::::::
18479 // compiler/init_attribute.h
```

```
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
optional<name> get_init_fn_name_for(environment const &env, name const &n);
inline bool has_init_attribute(environment const &env, name const &n) {
    return static_cast<bool>(get_init_fn_name_for(env, n));
}
}  // namespace lean
// ::::::::::::::
// compiler/init_module.h
// ::::::::::::::
/*
Copyright (c) 2015 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
namespace lean {
void initialize_compiler_module();
void finalize_compiler_module();
}  // namespace lean
// ::::::::::::::
// compiler/ir.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

#include "kernel/environment.h"
#include "library/compiler/util.h"
namespace lean {
namespace ir {
/*
inductive IRType
| float | uint8 | uint16 | uint32 | uint64 | usize
| irrelevant | object | tobject
| struct (leanTypeName : Option Name) (types : Array IRType) : IRType
| union (leanTypeName : Name) (types : Array IRType) : IRType

Remark: we don't create struct/union types from C++.
*/
enum class type {
    Float,
    UInt8,
    UInt16,
    UInt32,
    UInt64,
    USize,
    Irrelevant,
    Object,
    TObject
};

typedef nat var_id;
typedef nat jp_id;
typedef name fun_id;
```

```
18550 typedef object_ref arg;
18551 typedef object_ref expr;
18552 typedef object_ref param;
18553 typedef object_ref fn_body;
18554 typedef object_ref alt;
18555 typedef object_ref decl;
18556
18557 typedef object_ref decl;
18558 std::string decl_to_string(decl const &d);
18559 void test(decl const &d);
18560 environment compile(environment const &env, options const &opts,
18561                     comp_decls const &decls);
18562 environment add_extern(environment const &env, name const &fn);
18563 string_ref emit_c(environment const &env, name const &mod_name);
18564 }  // namespace ir
18565 void initialize_ir();
18566 void finalize_ir();
18567 }  // namespace lean
18568 // ::::::::::::::
18569 // compiler/ir_interpreter.h
18570 // ::::::::::::::
18571 /*
18572 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
18573 Released under Apache 2.0 license as described in the file LICENSE.
18574
18575 Author: Sebastian Ullrich
18576 */
18577 #pragma once
18578 #include <lean/object.h>
18579
18580 #include "kernel/environment.h"
18581
18582 namespace lean {
18583 namespace ir {
18584 /** \brief Run `n` using the "boxed" ABI, i.e. with all-owned parameters. */
18585 object *run_boxed(environment const &env, options const &opts, name const &fn,
18586                   unsigned n, object **args);
18587 uint32 run_main(environment const &env, options const &opts, int argv,
18588                 char *argc[]);
18589 }  // namespace ir
18590 void initialize_ir_interpreter();
18591 void finalize_ir_interpreter();
18592 }  // namespace lean
18593 // ::::::::::::::
18594 // compiler/lambda_lifting.h
18595 // ::::::::::::::
18596 /*
18597 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18598 Released under Apache 2.0 license as described in the file LICENSE.
18599
18600 Author: Leonardo de Moura
18601 */
18602 #pragma once
18603 #include "kernel/environment.h"
18604 #include "library/compiler/util.h"
18605 namespace lean {
18606 /** \brief Lift lambda expressions in `ds`. The result also contains new
18607  * auxiliary declarations that have been generated. */
18608 pair<environment, comp_decls> lambda_lifting(environment env,
18609                                              comp_decls const &ds);
18610 /* Return true iff `fn` is the name of an auxiliary function generated by
18611  * `lambda_lifting`. */
18612 bool is_lambda_lifting_name(name fn);
18613 };  // namespace lean
18614 // ::::::::::::::
18615 // compiler/lcnf.h
18616 // ::::::::::::::
18617 /*
18618 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18619 Released under Apache 2.0 license as described in the file LICENSE.
```

```
18620
18621 Author: Leonardo de Moura
18622 */
18623 #pragma once
18624 #include "kernel/environment.h"
18625 namespace lean {
18626 expr to_lcnf_core(environment const &env, local_ctx const &lctx, expr const &e);
18627 inline expr to_lcnf(environment const &env, expr const &e) {
18628     return to_lcnf_core(env, local_ctx(), e);
18629 }
18630 void initialize_lcnf();
18631 void finalize_lcnf();
18632 }  // namespace lean
18633 // ::::::::::::::
18634 // compiler/ll_infer_type.h
18635 // ::::::::::::::
18636 /*
18637 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18638 Released under Apache 2.0 license as described in the file LICENSE.
18639
18640 Author: Leonardo de Moura
18641 */
18642 #pragma once
18643 #include "kernel/environment.h"
18644 namespace lean {
18645 expr ll_infer_type(environment const &env, local_ctx const &lctx,
18646                    expr const &e);
18647 inline expr ll_infer_type(environment const &env, expr const &e) {
18648     return ll_infer_type(env, local_ctx(), e);
18649 }
18650 void ll_infer_type(environment const &env, comp_decls const &ds,
18651                    buffer<expr> &ts);
18652 void initialize_ll_infer_type();
18653 void finalize_ll_infer_type();
18654 }  // namespace lean
18655 // ::::::::::::::
18656 // compiler/llnf.h
18657 // ::::::::::::::
18658 /*
18659 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18660 Released under Apache 2.0 license as described in the file LICENSE.
18661
18662 Author: Leonardo de Moura
18663 */
18664 #pragma once
18665 #include "kernel/environment.h"
18666 #include "library/compiler/util.h"
18667
18668 namespace lean {
18669 environment compile_ir(environment const &env, options const &opts,
18670                        comp_decls const &ds);
18671
18672 bool is_llnf_apply(expr const &e);
18673 bool is_llnf_closure(expr const &e);
18674 bool is_llnf_cnstr(expr const &e, name &I, unsigned &cidx, unsigned &nusize,
18675                    unsigned &ssz);
18676 inline bool is_llnf_cnstr(expr const &e, unsigned &cidx, unsigned &nusize,
18677                           unsigned &ssz) {
18678     name I;
18679     return is_llnf_cnstr(e, I, cidx, nusize, ssz);
18680 }
18681 bool is_llnf_reuse(expr const &e, unsigned &cidx, unsigned &nusize,
18682                    unsigned &ssz, bool &updt_cidx);
18683 bool is_llnf_reset(expr const &e, unsigned &n);
18684 bool is_llnf_proj(expr const &e, unsigned &idx);
18685 bool is_llnf_sproj(expr const &e, unsigned &sz, unsigned &n, unsigned &offset);
18686 bool is_llnf_fproj(expr const &e, unsigned &n, unsigned &offset);
18687 bool is_llnf_uproj(expr const &e, unsigned &idx);
18688 bool is_llnf_sset(expr const &e, unsigned &sz, unsigned &n, unsigned &offset);
18689 bool is_llnf_fset(expr const &e, unsigned &n, unsigned &offset);
```

```
18690 bool is_llnf_uset(expr const &e, unsigned &n);
18691 bool is_llnf_jmp(expr const &e);
18692 bool is_llnf_unbox(expr const &e, unsigned &n);
18693 bool is_llnf_box(expr const &e, unsigned &n);
18694 bool is_llnf_inc(expr const &e);
18695 bool is_llnf_dec(expr const &e);
18696
18697 bool is_llnf_op(expr const &e);
18698 inline bool is_llnf_cnstr(expr const &e) {
18699     unsigned d1, d2, d3;
18700     return is_llnf_cnstr(e, d1, d2, d3);
18701 }
18702 inline bool is_llnf_reuse(expr const &e) {
18703     unsigned d1, d2, d3;
18704     bool u;
18705     return is_llnf_reuse(e, d1, d2, d3, u);
18706 }
18707 inline bool is_llnf_reset(expr const &e) {
18708     unsigned i;
18709     return is_llnf_reset(e, i);
18710 }
18711 inline bool is_llnf_proj(expr const &e) {
18712     unsigned d;
18713     return is_llnf_proj(e, d);
18714 }
18715 inline bool is_llnf_sproj(expr const &e) {
18716     unsigned d1, d2, d3;
18717     return is_llnf_sproj(e, d1, d2, d3);
18718 }
18719 inline bool is_llnf_fproj(expr const &e) {
18720     unsigned d1, d2;
18721     return is_llnf_fproj(e, d1, d2);
18722 }
18723 inline bool is_llnf_uproj(expr const &e) {
18724     unsigned d;
18725     return is_llnf_uproj(e, d);
18726 }
18727 inline bool is_llnf_sset(expr const &e) {
18728     unsigned d1, d2, d3;
18729     return is_llnf_sset(e, d1, d2, d3);
18730 }
18731 inline bool is_llnf_fset(expr const &e) {
18732     unsigned d1, d2;
18733     return is_llnf_fset(e, d1, d2);
18734 }
18735 inline bool is_llnf_uset(expr const &e) {
18736     unsigned d;
18737     return is_llnf_uset(e, d);
18738 }
18739 inline bool is_llnf_box(expr const &e) {
18740     unsigned n;
18741     return is_llnf_box(e, n);
18742 }
18743 inline bool is_llnf_unbox(expr const &e) {
18744     unsigned n;
18745     return is_llnf_unbox(e, n);
18746 }
18747
18748 expr get_constant_ll_type(environment const &env, name const &c);
18749 unsigned get_llnf_arity(environment const &env, name const &c);
18750
18751 struct field_info {
18752     /* Remark: the position of a scalar value in
18753        a constructor object is: `sizeof(void*)*m_idx + m_offset` */
18754     enum kind { Irrelevant, Object, USize, Scalar };
18755     kind m_kind;
18756     unsigned m_size;  // it is used only if `m_kind == Scalar`
18757     unsigned m_idx;
18758     unsigned m_offset;  // it is used only if `m_kind == Scalar`
18759     expr m_type;
```

```
18760     field_info() : m_kind(Irrelevant), m_idx(0), m_type(mk_enf_neutral()) {}
18761     field_info(unsigned idx)
18762         : m_kind(Object), m_idx(idx), m_type(mk_enf_object_type()) {}
18763     field_info(unsigned num, bool)
18764         : m_kind(USize), m_idx(num), m_type(mk_constant(get_usize_name())) {}
18765     field_info(unsigned sz, unsigned num, unsigned offset, expr const &type)
18766         : m_kind(Scalar),
18767           m_size(sz),
18768           m_idx(num),
18769           m_offset(offset),
18770           m_type(type) {}
18771     expr get_type() const { return m_type; }
18772     bool is_float() const { return is_constant(m_type, get_float_name()); }
18773     static field_info mk_irrelevant() { return field_info(); }
18774     static field_info mk_object(unsigned idx) { return field_info(idx); }
18775     static field_info mk_usize() { return field_info(0, true); }
18776     static field_info mk_scalar(unsigned sz, expr const &type) {
18777         return field_info(sz, 0, 0, type);
18778     }
18779 };
18780
18781 struct cnstr_info {
18782     unsigned m_cidx;
18783     list<field_info> m_field_info;
18784     unsigned m_num_objs{0};
18785     unsigned m_num_usizes{0};
18786     unsigned m_scalar_sz{0};
18787     cnstr_info(unsigned cidx, list<field_info> const &finfo);
18788 };
18789
18790 cnstr_info get_cnstr_info(type_checker::state &st, name const &n);
18791
18792 void initialize_llnf();
18793 void finalize_llnf();
18794 }  // namespace lean
18795 // ::::::::::::::
18796 // compiler/reduce_arity.h
18797 // ::::::::::::::
18798 /*
18799 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18800 Released under Apache 2.0 license as described in the file LICENSE.
18801 Author: Leonardo de Moura
18802 */
18803 #pragma once
18804 #include "library/compiler/util.h"
18805 namespace lean {
18806 comp_decls reduce_arity(environment const &env, comp_decls const &cdecls);
18807 /* Return true if the `cdecl` is of the form `f := fun xs, f._rarg ...`.
18808    That is, `f`s arity "was reduced" and an auxiliary declaration `f._rarg` was
18809    created to replace it. */
18810 bool arity_was_reduced(comp_decl const &cdecl);
18811 }  // namespace lean
18812 // ::::::::::::::
18813 // compiler/simp_app_args.h
18814 // ::::::::::::::
18815 /*
18816 Copyright (c) 2018 Microsoft Corporation. All rights reserved.
18817 Released under Apache 2.0 license as described in the file LICENSE.
18818
18819 Author: Leonardo de Moura
18820 */
18821 #pragma once
18822 #include "kernel/environment.h"
18823 namespace lean {
18824 expr simp_app_args(environment const &env, expr const &e);
18825 }
18826 // ::::::::::::::
18827 // compiler/specialize.h
18828 // ::::::::::::::
18829 /*
```

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

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/environment.h"
#include "library/compiler/csimp.h"
#include "library/compiler/util.h"
namespace lean {
pair<environment, comp_decls> specialize(environment env, comp_decls const &ds,
                                         csimp_cfg const &cfg);
void initialize_specialize();
void finalize_specialize();
}  // namespace lean
// ::::::::::::::
// compiler/struct_cases_on.h
// ::::::::::::::
/*
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
/* Insert `S.casesOn` applications for a structure `S` when
   1- There is a constructor application `S.mk ... x ...`, and
   2- `x := y.i`, and
   3- There is no `S.casesOn y ...`

   This transformation is useful because the `reset/reuse` insertion
   procedure uses `casesOn` applications as a guide.
   Moreover, Lean structure update expressions are not compiled using
   `casesOn` applicactions.

   Example: given
   ```
   fun x,
   let y_1 := x.1 in
   let y_2 := 0 in
   (y_1, y_2)
   ```
   this function returns
   ```
   fun x,
   Prod.casesOn x
    (fun fst snd,
      let y_1 := x.1 in
      let y_2 := 0 in
      (y_1, y_2))
   ```
   Note that, we rely on the simplifier (csimp.cpp) to replace `x.1` with `fst`.

   Remark: this function assumes we have already erased irrelevant information.

   Remark: we have considered compiling the `{ x with ... }` expressions using
   `casesOn`, but we loose useful definitional equalities. In the encoding we
   use,
   `{x with field1 := v1, field2 := v2}.field1` is definitional equal to `v1`.
   If we compile this expression using `casesOn`, we would have
   ```
   (match x with
    | {field1 := _, field2 := _, field3 := v3} := {field1 := v1, field2 := v2,
   field3 := v3}).field1
   ```
   as is only definitionally equal to `v1` IF `x` is definitionally equal to a
   constructor application. The missing definitional equalities is problematic.
```

```
   For example, the whole algebraic hierarchy in Lean relies on them.
*/
expr struct_cases_on(environment const &env, expr const &e);
}  // namespace lean
// ::::::::::::::
// compiler/util.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Author: Leonardo de Moura
*/
#pragma once
#include "kernel/expr.h"
#include "kernel/type_checker.h"
#include "library/constants.h"
#include "library/util.h"
#include "util/list_ref.h"
#include "util/name_hash_set.h"
#include "util/pair_ref.h"

namespace lean {
/* Return the `some(n)` if `I` is the name of an inductive datatype that
   contains only constructors with 0-arguments, and `n` is `1`, `2` or `4`,
   i.e., the number of bytes that should be used to store a value of this type.
   Otherwise, it return `none`.
   Remark: if the inductive datatype `I` has more than `2^32` constructors (very
   unlikely), the result is also `none`. */
optional<unsigned> is_enum_type(environment const &env, name const &I);

optional<expr> to_uint_type(unsigned nbytes);

/* A "compiler" declaration `x := e` */
typedef pair_ref<name, expr> comp_decl;
typedef list_ref<comp_decl> comp_decls;

void trace_comp_decl(comp_decl const &d);
void trace_comp_decls(comp_decls const &ds);

unsigned get_num_nested_lambdas(expr e);

bool is_lcnf_atom(expr const &e);

expr elim_trivial_let_decls(expr const &e);

bool has_inline_attribute(environment const &env, name const &n);
bool has_noinline_attribute(environment const &env, name const &n);
bool has_inline_if_reduce_attribute(environment const &env, name const &n);
bool has_never_extract_attribute(environment const &env, name const &n);

expr unfold_macro_defs(environment const &env, expr const &e);

/* Return true if the given argument is mdata relevant to the compiler

   Remark: we currently don't keep any metadata in the compiler. */
inline bool is_lc_mdata(expr const &) { return false; }

bool is_cases_on_recursor(environment const &env, name const &n);
/* We defined the "arity" of a cases_on application as the sum:
   ```
     number of inductive parameters +
     1 +   // motive
     number of inductive indices +
     1 +   // major premise
     number of constructors // cases_on has a minor premise for each constructor
   ```
   \pre is_cases_on_recursor(env, c) */
unsigned get_cases_on_arity(environment const &env, name const &c,
                            bool before_erasure = true);
```

```
/* Return the `inductive_val` for the cases_on constant `c`. */
inline inductive_val get_cases_on_inductive_val(environment const &env,
                                                name const &c) {
    lean_assert(is_cases_on_recursor(env, c));
    return env.get(c.get_prefix()).to_inductive_val();
}
inline inductive_val get_cases_on_inductive_val(environment const &env,
                                                expr const &c) {
    lean_assert(is_constant(c));
    return get_cases_on_inductive_val(env, const_name(c));
}
inline bool is_cases_on_app(environment const &env, expr const &e) {
    expr const &fn = get_app_fn(e);
    return is_constant(fn) && is_cases_on_recursor(env, const_name(fn));
}
/* Return the major premise of a cases_on-application.
   \pre is_cases_on_app(env, c) */
expr get_cases_on_app_major(environment const &env, expr const &c,
                            bool before_erasure = true);
unsigned get_cases_on_major_idx(environment const &env, name const &c,
                                bool before_erasure = true);
/* Return the pair `(b, e)` such that `i in [b, e)` is argument `i` in a `c`
   cases_on application is a minor premise. \pre is_cases_on_recursor(env, c) */
pair<unsigned, unsigned> get_cases_on_minors_range(environment const &env,
                                                   name const &c,
                                                   bool before_erasure = true);

inline bool is_quot_primitive(environment const &env, name const &n) {
    optional<constant_info> info = env.find(n);
    return info && info->is_quot();
}

inline bool is_lc_unreachable_app(expr const &e) {
    return is_app_of(e, get_lc_unreachable_name(), 1);
}
inline bool is_lc_proof_app(expr const &e) {
    return is_app_of(e, get_lc_proof_name(), 1);
}

expr mk_lc_unreachable(type_checker::state &s, local_ctx const &lctx,
                       expr const &type);

inline name mk_join_point_name(name const &n) { return name(n, "_join"); }
bool is_join_point_name(name const &n);
/* Pseudo "do" joinpoints are used to implement a temporary HACK. See
 * `visit_let` method at `lcnf.cpp` */
inline name mk_pseudo_do_join_point_name(name const &n) {
    return name(n, "_do_jp");
}
bool is_pseudo_do_join_point_name(name const &n);

/* Create an auxiliary names for a declaration that saves the result of the
   compilation after step simplification. */
inline name mk_cstage1_name(name const &decl_name) {
    return name(decl_name, "_cstage1");
}

/* Create an auxiliary names for a declaration that saves the result of the
   compilation after step erasure. */
inline name mk_cstage2_name(name const &decl_name) {
    return name(decl_name, "_cstage2");
}

/* Set `used[i] = true` if `fvars[i]` occurs in `e` */
void mark_used_fvars(expr const &e, buffer<expr> const &fvars,
                     buffer<bool> &used);

/* Return true if `e` contains the free variable `fvar` */
bool has_fvar(expr const &e, expr const &fvar);

```

```
19040 expr replace_fvar(expr const &e, expr const &fvar, expr const &new_term);
19041
19042 void sort_fvars(local_ctx const &lctx, buffer<expr> &fvars);
19043
19044 /* Return the "code" size for `e` */
19045 unsigned get_lcnf_size(environment const &env, expr e);
19046
19047 // ========================================
19048 // Auxiliary expressions for erasure.
19049 // We use them after we have erased proofs and unnecessary type information.
19050 // `enf` stands for "erasure normal form". It is LCNF after erasure.
19051
19052 /* Create a neutral expression used at ENF */
19053 expr mk_enf_neutral();
19054 /* Create an unreachable expression used at ENF */
19055 expr mk_enf_unreachable();
19056 expr mk_enf_object_type();
19057 expr mk_enf_neutral_type();
19058 /* "Void" type used in LLNF. Remark: the ENF types neutral and object are also
19059  * used in LLNF. */
19060 expr mk_llnf_void_type();
19061
19062 bool is_enf_neutral(expr const &e);
19063 bool is_enf_unreachable(expr const &e);
19064 bool is_enf_object_type(expr const &e);
19065 bool is_llnf_void_type(expr const &e);
19066 bool is_llnf_unboxed_type(expr const &type);
19067
19068 /* Return (some idx) iff inductive datatype `I_name` is safe, has only one
19069    constructor, and this constructor has only one relevant field, `idx` is the
19070    field position. */
19071 optional<unsigned> has_trivial_structure(environment const &env,
19072                                          name const &I_name);
19073
19074 expr mk_runtime_type(type_checker::state &st, local_ctx const &lctx, expr e);
19075
19076 // ========================================
19077
19078 /* Return true if `n` is the name of a type with builtin support in the code
19079  * generator. */
19080 bool is_runtime_builtin_type(name const &n);
19081 inline bool is_runtime_builtin_type(expr const &e) {
19082     return is_constant(e) && is_runtime_builtin_type(const_name(e));
19083 }
19084
19085 /* Return true if `n` is the name of a type that is treated as a scalar type by
19086  * the code generator. */
19087 bool is_runtime_scalar_type(name const &n);
19088
19089 bool is_irrelevant_type(type_checker::state &st, local_ctx lctx,
19090                         expr const &type);
19091 bool is_irrelevant_type(environment const &env, expr const &type);
19092
19093 void collect_used(expr const &e, name_hash_set &S);
19094 /* Return true iff `e` contains a free variable in `s` */
19095 bool depends_on(expr const &e, name_hash_set const &s);
19096
19097 bool is_stage2_decl(environment const &env, name const &n);
19098 environment register_stage1_decl(environment const &env, name const &n,
19099                                  names const &ls, expr const &t, expr const &v);
19100 environment register_stage2_decl(environment const &env, name const &n,
19101                                  expr const &t, expr const &v);
19102
19103 /* Return `some n` iff `e` is of the forms `expr.lit (literal.nat n)` or
19104  * `uint*.of_nat (expr.lit (literal.nat n))` */
19105 optional<nat> get_num_lit_ext(expr const &e);
19106 inline bool is_morally_num_lit(expr const &e) {
19107     return static_cast<bool>(get_num_lit_ext(e));
19108 }
19109
```

```
/* Return `some n` if `c` is of the form `fix_core_n` where `n in [1, 6]`.
   Remark: this function is assuming the core library contains `fix_core_1` ...
   `fix_core_6` definitions. */
optional<unsigned> is_fix_core(name const &c);
/* Return the `fix_core_n` constant, and `none` if `n` is not in `[1, 6]`.
   Remark: this function is assuming the core library contains `fix_core_1` ...
   `fix_core_6` definitions.
   Remark: this function assumes universe levels have already been erased. */
optional<expr> mk_enf_fix_core(unsigned n);

bool lcnf_check_let_decls(environment const &env, comp_decl const &d);
bool lcnf_check_let_decls(environment const &env, comp_decls const &ds);

// =======================================
/* Similar to `type_checker::eta_expand`, but preserves LCNF */
expr lcnf_eta_expand(type_checker::state &st, local_ctx lctx, expr e);

// =======================================
// UInt and USize helper functions

expr mk_usize_type();
bool is_usize_type(expr const &e);
optional<unsigned> is_builtin_scalar(expr const &type);
optional<unsigned> is_enum_type(environment const &env, expr const &type);

// =======================================

void initialize_compiler_util();
void finalize_compiler_util();
}  // namespace lean
// ::::::::::::::
// constructions/brec_on.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
/** \brief Given an inductive datatype \c n in \c env, add
    <tt>n.below</tt> auxiliary construction for <tt>n.brec_on</t>
    (aka below recursion on) to the environment.
*/
environment old_mk_below(environment const &env, name const &n);
environment old_mk_ibelow(environment const &env, name const &n);

environment old_mk_brec_on(environment const &env, name const &n);
environment old_mk_binduction_on(environment const &env, name const &n);

environment mk_below(environment const &env, name const &n);
environment mk_ibelow(environment const &env, name const &n);

environment mk_brec_on(environment const &env, name const &n);
environment mk_binduction_on(environment const &env, name const &n);
}  // namespace lean
// ::::::::::::::
// constructions/cases_on.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

```
19180
19181 namespace lean {
19182 /** \brief Given an inductive datatype \c n in \c env, add
19183     <tt>n.cases_on</tt> to the environment.
19184
19185     \remark Throws an exception if \c n is not an inductive datatype.
19186 */
19187 environment mk_cases_on(environment const &env, name const &n);
19188 }  // namespace lean
19189 // ::::::::::::::
19190 // constructions/init_module.h
19191 // ::::::::::::::
19192 /*
19193 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
19194 Released under Apache 2.0 license as described in the file LICENSE.
19195
19196 Author: Leonardo de Moura
19197 */
19198 #pragma once
19199
19200 namespace lean {
19201 void initialize_constructions_module();
19202 void finalize_constructions_module();
19203 }  // namespace lean
19204 // ::::::::::::::
19205 // constructions/no_confusion.h
19206 // ::::::::::::::
19207 /*
19208 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
19209 Released under Apache 2.0 license as described in the file LICENSE.
19210
19211 Author: Leonardo de Moura
19212 */
19213 #pragma once
19214 #include "kernel/environment.h"
19215
19216 namespace lean {
19217 /** \brief Given an inductive datatype \c n (which is not a proposition) in \c
19218    env, add <tt>n.no_confusion_type</tt> and <tt>n.no_confusion</tt> to the
19219    environment.
19220
19221     \remark This procedure assumes the environment contains eq, n.cases_on</tt>.
19222     If the environment has an impredicative Prop, it also assumes heq is
19223    defined. If the environment does not have an impredicative Prop, then it also
19224    assumes lift is defined.
19225 */
19226 environment mk_no_confusion(environment const &env, name const &n);
19227 }  // namespace lean
19228 // ::::::::::::::
19229 // constructions/projection.h
19230 // ::::::::::::::
19231 /*
19232 Copyright (c) 2014 Microsoft Corporation. All rights reserved.
19233 Released under Apache 2.0 license as described in the file LICENSE.
19234
19235 Author: Leonardo de Moura
19236 */
19237 #pragma once
19238 #include "kernel/environment.h"
19239 #include "library/util.h"
19240
19241 namespace lean {
19242 /** \brief Create projections operators for the structure named \c n.
19243     The procedure assumes \c n is a structure.
19244
19245     The argument \c infer_kinds specifies the implicit argument inference
19246    strategies used for the structure parameters.
19247
19248     If \c inst_implicit == true, then the structure argument of the projection
19249    is decorated as "instance implicit" [s : n]
```

```
*/
environment mk_projections(environment const &env, name const &n,
                           buffer<name> const &proj_names,
                           buffer<implicit_infer_kind> const &infer_kinds,
                           bool inst_implicit = false);

void initialize_def_projection();
void finalize_def_projection();
}  // namespace lean
// ::::::::::::::
// constructions/rec_on.h
// ::::::::::::::
/*
Copyright (c) 2014 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
/** \brief Given an inductive datatype \c n in \c env, add
    <tt>n.rec_on</tt> to the environment.

    \remark <tt>rec_on</tt> is based on <tt>n.rec</tt>

    \remark Throws an exception if \c n is not an inductive datatype.
*/
environment mk_rec_on(environment const &env, name const &n);
}  // namespace lean
// ::::::::::::::
// constructions/util.h
// ::::::::::::::
/*
Copyright (c) 2018 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

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

namespace lean {
environment completion_add_to_black_list(environment const &env,
                                         name const &decl_name);

expr mk_pprod(type_checker &ctx, expr const &a, expr const &b, bool prop);
expr mk_pprod_mk(type_checker &ctx, expr const &a, expr const &b, bool prop);
expr mk_pprod_fst(type_checker &ctx, expr const &p, bool prop);
expr mk_pprod_snd(type_checker &ctx, expr const &p, bool prop);

name_generator mk_constructions_name_generator();
void initialize_constructions_util();
void finalize_constructions_util();
}  // namespace lean
```