



```
2133         fp->ctf_flags |= LCTF_DIRTY;
2135     return (0);
2136 }
unchanged portion omitted
```

new/usr/src/lib/libctf/common/ctf\_convert.c

```
*****  
4307 Tue Apr 23 05:24:01 2019  
new/usr/src/lib/libctf/common/ctf_convert.c  
10812 ctf tools shouldn't add blank labels  
10813 ctf symbol mapping needs work  
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>  
*****  
1 /*  
2  * This file and its contents are supplied under the terms of the  
3  * Common Development and Distribution License (" CDDL"), version 1.0.  
4  * You may only use this file in accordance with the terms of version  
5  * 1.0 of the CDDL.  
6  *  
7  * A full copy of the text of the CDDL should have accompanied this  
8  * source. A copy of the CDDL is also available via the Internet at  
9  * http://www.illumos.org/license/CDDL.  
10 */  
12 /*  
13  * Copyright 2019 Joyent, Inc.  
14  * Copyright 2015 Joyent, Inc.  
14 */  
16 /*  
17  * Main conversion entry points. This has been designed such that there can be  
18  * any number of different conversion backends. Currently we only have one that  
19  * understands DWARFv2 (and bits of DWARFv4). Each backend should be placed in  
20  * the ctf_converters list and each will be tried in turn.  
21 */  
23 #include <libctf_impl.h>  
24 #include <elf.h>  
26 ctf_convert_f ctf_converters[] = {  
27     ctf_dwarf_convert  
28 };  
_____unchanged_portion_omitted_____  
104 static ctf_file_t *  
105 ctf_elfconvert(int fd, Elf *elf, const char *label, uint_t nthrs, uint_t flags,  
106     int *errp, char *errbuf, size_t errlen)  
107 {  
108     int err, i;  
109     ctf_file_t *fp = NULL;  
110     boolean_t notsup = B_TRUE;  
111     ctf_convert_source_t type;  
112  
113     if (*errp == NULL)  
114         errp = &err;  
115  
116     if (elf == NULL) {  
117         *errp = EINVAL;  
118         return (NULL);  
119     }  
120  
121     if (flags & ~CTF_CONVERT_F_IGNNNONC) {  
122         *errp = EINVAL;  
123         return (NULL);  
124     }  
125  
126     if (elf_kind(elf) != ELF_K_ELF) {  
127         *errp = ECTF_FMT;  
128         return (NULL);  
129     }  
130  
131     ctf_convert_ftypes(elf, &type);
```

1

new/usr/src/lib/libctf/common/ctf\_convert.c

```
132     ctf_dprintf("got types: %d\n", type);  
133     if (flags & CTF_CONVERT_F_IGNNNONC) {  
134         if (type == CTFCONV_SOURCE_NONE ||  
135             (type & CTFCONV_SOURCE_UNKNOWN)) {  
136             *errp = ECTF_CONVNOSRC;  
137             return (NULL);  
138         }  
139     }  
140     for (i = 0; i < NCONVERTS; i++) {  
141         ctf_conv_status_t cs;  
142  
143         fp = NULL;  
144         cs = ctf_converters[i](fd, elf, nthrs, errp, &fp, errbuf,  
145             errlen);  
146         if (cs == CTF_CONV_SUCCESS) {  
147             notsup = B_FALSE;  
148             break;  
149         }  
150         if (cs == CTF_CONV_ERROR) {  
151             fp = NULL;  
152             notsup = B_FALSE;  
153             break;  
154         }  
155     }  
156     if (notsup == B_TRUE) {  
157         if ((flags & CTF_CONVERT_F_IGNNNONC) != 0 &&  
158             (type & CTFCONV_SOURCE_C) == 0) {  
159             *errp = ECTF_CONVNOSRC;  
160             return (NULL);  
161         }  
162         *errp = ECTF_NOCONVBKEND;  
163         return (NULL);  
164     }  
165     /*  
166      * Succsesful conversion.  
167      */  
168     if (fp != NULL && label != NULL) {  
169         if (fp != NULL) {  
170             if (label == NULL)  
171                 label = "";  
172             if (ctf_add_label(fp, label, fp->ctf_typemax, 0) == CTF_ERR) {  
173                 *errp = ctf_errno(fp);  
174                 ctf_close(fp);  
175                 return (NULL);  
176             }  
177             if (ctf_update(fp) == CTF_ERR) {  
178                 *errp = ctf_errno(fp);  
179                 ctf_close(fp);  
180                 return (NULL);  
181             }  
182         }  
183         return (fp);  
184     }  
185 }  
_____unchanged_portion_omitted_____
```

2

```
*****
82711 Tue Apr 23 05:24:01 2019
new/usr/src/lib/libctf/common/ctf_dwarf.c
10812 ctf tools shouldn't add blank labels
10813 ctf symbol mapping needs work
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
*****
```

```

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /*
26 * Copyright 2012 Jason King. All rights reserved.
27 * Use is subject to license terms.
28 */

30 /*
31 * Copyright 2019 Joyent, Inc.
31 * Copyright 2018 Joyent, Inc.
32 */

34 /*
35 * CTF DWARF conversion theory.
36 *
37 * DWARF data contains a series of compilation units. Each compilation unit
38 * generally refers to an object file or what once was, in the case of linked
39 * binaries and shared objects. Each compilation unit has a series of what DWARF
40 * calls a DIE (Debugging Information Entry). The set of entries that we care
41 * about have type information stored in a series of attributes. Each DIE also
42 * has a tag that identifies the kind of attributes that it has.
43 *
44 * A given DIE may itself have children. For example, a DIE that represents a
45 * structure has children which represent members. Whenever we encounter a DIE
46 * that has children or other values or types associated with it, we recursively
47 * process those children first so that way we can then refer to the generated
48 * CTF type id while processing its parent. This reduces the amount of unknowns
49 * and fixups that we need. It also ensures that we don't accidentally add types
50 * that an overzealous compiler might add to the DWARF data but aren't used by
51 * anything in the system.
52 *
53 * Once we do a conversion, we store a mapping in an AVL tree that goes from the
54 * DWARF's die offset, which is relative to the given compilation unit, to a
55 * ctf_id_t.
56 *
57 * Unfortunately, some compilers actually will emit duplicate entries for a
58 * given type that look similar, but aren't quite. To that end, we go through
```

```

59 * and do a variant on a merge once we're done processing a single compilation
60 * unit which deduplicates all of the types that are in the unit.
61 *
62 * Finally, if we encounter an object that has multiple compilation units, then
63 * we'll convert all of the compilation units separately and then do a merge, so
64 * that way we can result in one single ctf_file_t that represents everything
65 * for the object.
66 *
67 * Conversion Steps
68 * -----
69 *
70 * Because a given object we've been given to convert may have multiple
71 * compilation units, we break the work into two halves. The first half
72 * processes each compilation unit (potentially in parallel) and then the second
73 * half optionally merges all of the dies in the first half. First, we'll cover
74 * what's involved in converting a single ctf_cu_t's dwarf to CTF. This covers
75 * the work done in ctf_dwarf_convert_one().
76 *
77 * An individual ctf_cu_t, which represents a compilation unit, is converted to
78 * CTF in a series of multiple passes.
79 *
80 * Pass 1: During the first pass we walk all of the top-level dies and if we
81 * find a function, variable, struct, union, enum or typedef, we recursively
82 * transform all of its types. We don't recurse or process everything, because
83 * we don't want to add some of the types that compilers may add which are
84 * effectively unused.
85 *
86 * During pass 1, if we encounter any structures or unions we mark them for
87 * fixing up later. This is necessary because we may not be able to determine
88 * the full size of a structure at the beginning of time. This will happen if
89 * the DWARF attribute DW_AT_byte_size is not present for a member. Because of
90 * this possibility we defer adding members to structures or even converting
91 * them during pass 1 and save that for pass 2. Adding all of the base
92 * structures without any of their members helps deal with any circular
93 * dependencies that we might encounter.
94 *
95 * Pass 2: This pass is used to do the first half of fixing up structures and
96 * unions. Rather than walk the entire type space again, we actually walk the
97 * list of structures and unions that we marked for later fixing up. Here, we
98 * iterate over every structure and add members to the underlying ctf_file_t,
99 * but not to the structs themselves. One might wonder why we don't, and the
100 * main reason is that libctf requires a ctf_update() be done before adding the
101 * members to structures or unions.
102 *
103 * Pass 3: This pass is used to do the second half of fixing up structures and
104 * unions. During this part we always go through and add members to structures
105 * and unions that we added to the container in the previous pass. In addition,
106 * we set the structure and union's actual size, which may have additional
107 * padding added by the compiler, it isn't simply the last offset. DWARF always
108 * guarantees an attribute exists for this. Importantly no ctf_id_t's change
109 * during pass 2.
110 *
111 * Pass 4: The next phase is to add CTF entries for all of the symbols and
112 * variables that are present in this die. During pass 1 we added entries to a
113 * map for each variable and function. During this pass, we iterate over the
114 * symbol table and when we encounter a symbol that we have in our lists of
115 * translated information which matches, we then add it to the ctf_file_t.
116 *
117 * Pass 5: Here we go and look for any weak symbols and functions and see if
118 * they match anything that we recognize. If so, then we add type information
119 * for them at this point based on the matching type.
120 *
121 * Pass 6: This pass is actually a variant on a merge. The traditional merge
122 * process expects there to be no duplicate types. As such, at the end of
123 * conversion, we do a dedup on all of the types in the system. The
124 * deduplication process is described in lib/libctf/common/ctf_merge.c.
```

```

125 *
126 * Once pass 6 is done, we've finished processing the individual compilation
127 * unit.
128 *
129 * The following steps reflect the general process of doing a conversion.
130 *
131 * 1) Walk the dwarf section and determine the number of compilation units
132 * 2) Create a ctf_cu_t for each compilation unit
133 * 3) Add all ctf_cu_t's to a workq
134 * 4) Have the workq process each die with ctf_dwarf_convert_one. This itself
135 * is comprised of several steps, which were already enumerated.
136 * 5) If we have multiple cu's, we do a ctf merge of all the dies. The mechanics
137 * of the merge are discussed in lib/libctf/common/ctf_merge.c.
138 * 6) Free everything up and return a ctf_file_t to the user. If we only had a
139 * single compilation unit, then we give that to the user. Otherwise, we
140 * return the merged ctf_file_t.
141 *
142 * Threading
143 *
144 *
145 * The process has been designed to be amenable to threading. Each compilation
146 * unit has its own type stream, therefore the logical place to divide and
147 * conquer is at the compilation unit. Each ctf_cu_t has been built to be able
148 * to be processed independently of the others. It has its own libdwarf handle,
149 * as a given libdwarf handle may only be used by a single thread at a time.
150 * This allows the various ctf_cu_t's to be processed in parallel by different
151 * threads.
152 *
153 * All of the ctf_cu_t's are loaded into a workq which allows for a number of
154 * threads to be specified and used as a thread pool to process all of the
155 * queued work. We set the number of threads to use in the workq equal to the
156 * number of threads that the user has specified.
157 *
158 * After all of the compilation units have been drained, we use the same number
159 * of threads when performing a merge of multiple compilation units, if they
160 * exist.
161 *
162 * While all of these different parts do support and allow for multiple threads,
163 * it's important that when only a single thread is specified, that it be the
164 * calling thread. This allows the conversion routines to be used in a context
165 * that doesn't allow additional threads, such as rtld.
166 *
167 * Common DWARF Mechanics and Notes
168 *
169 *
170 * At this time, we really only support DWARFv2, though support for DWARFv4 is
171 * mostly there. There is no intent to support DWARFv3.
172 *
173 * Generally types for something are stored in the DW_AT_type attribute. For
174 * example, a function's return type will be stored in the local DW_AT_type
175 * attribute while the arguments will be in child DIEs. There are also various
176 * times when we don't have any DW_AT_type. In that case, the lack of a type
177 * implies, at least for C, that its C type is void. Because DWARF doesn't emit
178 * one, we have a synthetic void type that we create and manipulate instead and
179 * pass it off to consumers on an as-needed basis. If nothing has a void type,
180 * it will not be emitted.
181 *
182 * Architecture Specific Parts
183 *
184 *
185 * The CTF tooling encodes various information about the various architectures
186 * in the system. Importantly, the tool assumes that every architecture has a
187 * data model where long and pointer are the same size. This is currently the
188 * case, as the two data models illumos supports are ILP32 and LP64.
189 *
190 * In addition, we encode the mapping of various floating point sizes to various

```

```

191 * types for each architecture. If a new architecture is being added, it should
192 * be added to the list. The general design of the ctf conversion tools is to be
193 * architecture independent. eg. any of the tools here should be able to convert
194 * any architecture's DWARF into ctf; however, this has not been rigorously
195 * tested and more importantly, the ctf routines don't currently write out the
196 * data in an endian-aware form, they only use that of the currently running
197 * library.
198 */

200 #include <libctf_impl.h>
201 #include <sys/avl.h>
202 #include <sys/debug.h>
203 #include <gelf.h>
204 #include <libdwarf.h>
205 #include <dwarf.h>
206 #include <libgen.h>
207 #include <workq.h>
208 #include <errno.h>

210 #define DWARF_VERSION_TWO          2
211 #define DWARF_VARARGS_NAME        "..."

213 /*
214 * Dwarf may refer recursively to other types that we've already processed. To
215 * see if we've already converted them, we look them up in an AVL tree that's
216 * sorted by the DWARF id.
217 */
218 typedef struct ctf_dwmap {
219     avl_node_t      cdm_avl;
220     Dwarf_Off       cdm_off;
221     Dwarf_Die      cdm_die;
222     ctf_id_t        cdm_id;
223     boolean_t       cdm_fix;
224 } ctf_dwmap_t;
225 unchanged portion omitted

276 static int ctf_dwarf_offset(ctf_cu_t *, Dwarf_Die, Dwarf_Off *);
277 static int ctf_dwarf_convert_die(ctf_cu_t *, Dwarf_Die);
278 static int ctf_dwarf_convert_type(ctf_cu_t *, Dwarf_Die, ctf_id_t *, int);

280 static int ctf_dwarf_function_count(ctf_cu_t *, Dwarf_Die, ctf_funcinfo_t *,
281                                     boolean_t);
282 static int ctf_dwarf_convert_fargs(ctf_cu_t *, Dwarf_Die, ctf_funcinfo_t *,
283                                   ctf_id_t *);

285 typedef int (ctf_dwarf_symtab_f)(ctf_cu_t *, const GElf_Sym *, ulong_t,
286                                 const char *, const char *, void *);
287
288 /*
289 * This is a generic way to set a CTF Conversion backend error depending on what
290 * we were doing. Unless it was one of a specific set of errors that don't
291 * indicate a programming / translation bug, eg. ENOMEM, then we transform it
292 * into a CTF backend error and fill in the error buffer.
293 */
294 static int
295 ctf_dwarf_error(ctf_cu_t *cup, ctf_file_t *cfp, int err, const char *fmt, ...)
296 {
297     va_list ap;
298     int ret;
299     size_t off = 0;
300     ssize_t rem = cup->cu_errlen;
301     if (cfp != NULL)
302         err = ctf_errno(cfp);
303
304     if (err == ENOMEM)
305         return (err);

```

```

304     ret = snprintf(cup->cu_errbuf, rem, "die %s: ", cup->cu_name);
305     if (ret < 0)
306         goto err;
307     off += ret;
308     rem = MAX(rem - ret, 0);

310     va_start(ap, fmt);
311     ret = vsnprintf(cup->cu_errbuf + off, rem, fmt, ap);
312     va_end(ap);
313     if (ret < 0)
314         goto err;

316     off += ret;
317     rem = MAX(rem - ret, 0);
318     if (fmt[strlen(fmt) - 1] != '\n') {
319         (void) snprintf(cup->cu_errbuf + off, rem,
320                         "%s\n", ctf_errmsg(err));
321     }
322     va_end(ap);
323     return (ECTF_CONVBKERR);

325 err:
326     cup->cu_errbuf[0] = '\0';
327     return (ECTF_CONVBKERR);
328 }

unchanged_portion_omitted

2166 /*
2167 * The DWARF information about a symbol and the information in the symbol table
2168 * may not be the same due to symbol reduction that is performed by ld due to a
2169 * mapfile or other such directive. We process weak symbols at a later time.
2170 */
2171 /* The following are the rules that we employ:
2172 */
2173 /* 1. A DWARF function that is considered exported matches STB_GLOBAL entries
2174 * with the same name.
2175 */
2176 /* 2. A DWARF function that is considered exported matches STB_LOCAL entries
2177 * with the same name and the same file. This case may happen due to mapfile
2178 * reduction.
2179 */
2180 /* 3. A DWARF function that is not considered exported matches STB_LOCAL entries
2181 * with the same name and the same file.
2182 */
2183 /* 4. A DWARF function that has the same name as the symbol table entry, but the
2184 * files do not match. This is considered a 'fuzzy' match. This may also happen
2185 * due to a mapfile reduction. Fuzzy matching is only used when we know that the
2186 * file in question refers to the primary object. This is because when a symbol
2187 * is reduced in a mapfile, it's always going to be tagged as a local value in
2188 * the generated output and it is considered as to belong to the primary file
2189 * which is the first STT_FILE symbol we see.
2190 */
2191 static boolean_t
2192 ctf_dwarf_symbol_match(const char *symtab_file, const char *symtab_name,
2193     uint_t symtab_bind, const char *dwarf_file, const char *dwarf_name,
2194     boolean_t dwarf_global, boolean_t *is_fuzzy)
2195 {
2196     *is_fuzzy = B_FALSE;

2198     if (symtab_bind != STB_LOCAL && symtab_bind != STB_GLOBAL) {
2199         return (B_FALSE);
2200     }

2202     if (strcmp(symtab_name, dwarf_name) != 0) {
2203         return (B_FALSE);

```

```

2204     }

2206     if (symtab_bind == STB_GLOBAL) {
2207         return (dwarf_global);
2208     }

2210     if (strcmp(symtab_file, dwarf_file) == 0) {
2211         return (B_TRUE);
2212     }

2214     if (dwarf_global) {
2215         *is_fuzzy = B_TRUE;
2216         return (B_TRUE);
2217     }

2219     return (B_FALSE);
2220 }

2222 static ctf_dwfunc_t *
2223 ctf_dwarf_match_func(ctf_cu_t *cup, const char *file, const char *name,
2224     uint_t bind, boolean_t primary)
2225     int bind)
2226 {
2227     ctf_dwfunc_t *cdf, *fuzzy = NULL;
2228     ctf_dwfunc_t *cdf;
2229

2230     if (bind == STB_WEAK)
2231         return (NULL);

2232     /* Nothing we can do if we can't find a name to compare it to. */
2233     if (bind == STB_LOCAL && (file == NULL || cup->cu_name == NULL))
2234         return (NULL);

2235     for (cdf = ctf_list_next(&cup->cu_funcs); cdf != NULL;
2236         cdf = ctf_list_next(cdf)) {
2237         boolean_t is_fuzzy = B_FALSE;

2238         if (ctf_dwarf_symbol_match(file, name, bind, cup->cu_name,
2239             cdf->cdf_name, cdf->cdf_global, &is_fuzzy)) {
2240             if (is_fuzzy) {
2241                 if (primary) {
2242                     fuzzy = cdf;
2243                 }
2244             if (bind == STB_GLOBAL && cdf->cdf_global == B_FALSE)
2245                 continue;
2246             } else {
2247                 if (bind == STB_LOCAL && cdf->cdf_global == B_TRUE)
2248                     continue;
2249                 if (strcmp(name, cdf->cdf_name) != 0)
2250                     continue;
2251                 if (bind == STB_LOCAL && strcmp(file, cup->cu_name) != 0)
2252                     continue;
2253             }
2254         }
2255     }

2256     return (fuzzy);
2257     return (NULL);
2258 }

2259 static ctf_dwvar_t *
2260 ctf_dwarf_match_var(ctf_cu_t *cup, const char *file, const char *name,
2261     uint_t bind, boolean_t primary)
2262     int bind)
2263 {

```

```

2258     ctf_dwarf_t *cdv, *fuzzy = NULL;
2261
2260     if (bind == STB_WEAK)
2261         return (NULL);
2263
2263 /* Nothing we can do if we can't find a name to compare it to.*/
2264     if (bind == STB_LOCAL && (file == NULL || cup->cu_name == NULL))
2265         return (NULL);
2266     ctf_dprintf("Still considering %s\n", name);
2267
2268     for (cdv = ctf_list_next(&cup->cu_vars); cdv != NULL;
2269          cdv = ctf_list_next(cdv)) {
2270         boolean_t is_fuzzy = B_FALSE;
2271         if (bind == STB_GLOBAL && cdv->cdv_global == B_FALSE)
2272             continue;
2273         if (bind == STB_LOCAL && cdv->cdv_global == B_TRUE)
2274             continue;
2275         if (strcmp(name, cdv->cdv_name) != 0)
2276             continue;
2277         if (bind == STB_LOCAL && strcmp(file, cup->cu_name) != 0)
2278             continue;
2279         return (cdv);
2280     }
2281
2282     if (ctf_dwarf_symbol_match(file, name, bind, cup->cu_name,
2283                               cdv->cdv_name, cdv->cdv_global, &is_fuzzy)) {
2284         if (is_fuzzy) {
2285             if (primary) {
2286                 fuzzy = cdv;
2287             }
2288         }
2289     }
2290
2291     return (NULL);
2292 }
2293
2294 static int
2295 ctf_dwarf_symtab_iter(ctf_cu_t *cup, ctf_dwarf_symtab_f *func, void *arg)
2296 {
2297     int ret;
2298     ulong_t i;
2299     ctf_file_t *fp = cup->cu_ctfp;
2300     const char *file = NULL;
2301     uintptr_t symbase = (uintptr_t)fp->ctf_symtab.cts_data;
2302     uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;
2303
2304     for (i = 0; i < fp->ctf_nsyms; i++) {
2305         const char *name;
2306         int type;
2307         GElf_Sym gsym;
2308         const GElf_Sym *gsymp;
2309
2310         if (fp->ctf_symtab.cts_entsize == sizeof(Elf32_Sym)) {
2311             const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
2312             type = ELF32_ST_TYPE(symp->st_info);
2313             if (type == STT_FILE) {
2314                 file = (char*)(strbase + symp->st_name);
2315                 continue;
2316             }
2317             if (type != STT_OBJECT && type != STT_FUNC)
2318                 continue;
2319             if (ctf_sym_valid(strbase, type, symp->st_shndx,
2320                               symp->st_value, symp->st_name) == B_FALSE)
2321                 continue;
2322             name = (char*)(strbase + symp->st_name);
2323             gsym.st_name = symp->st_name;
2324             gsym.st_value = symp->st_value;
2325             gsym.st_size = symp->st_size;
2326             gsym.st_info = symp->st_info;
2327         }
2328     }
2329
2330     if (ctf_dwarf_symbol_match(file, name, bind, name,
2331                               name, type, &ret)) {
2332         if (ret == B_TRUE)
2333             func(cup, gsymp, i, file, name, arg);
2334     }
2335
2336     return (ret);
2337 }
2338
2339 static int
2340 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2341                           const char *file, const char *name, boolean_t primary, void *arg)
2342 {
2343     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2344                               const char *file, const char *name, void *arg)
2345     {
2346         int ret;
2347         uint_t bind, type;
2348         ctf_cu_t *cup = arg;
2349         int ret, bind, type;
2350
2351         bind = GELF_ST_BIND(symp->st_info);
2352         type = GELF_ST_TYPE(symp->st_info);
2353
2354         /*
2355          * Come back to weak symbols in another pass
2356          */
2357         if (bind == STB_WEAK)
2358             return (0);
2359
2360         if (type == STT_OBJECT) {
2361             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2362                                                 bind, primary);
2363             bind;
2364             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2365             if (cdv == NULL)
2366                 return (0);
2367             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2368             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2369             ctf_dprintf("added object %s\n", name);
2370         } else {
2371             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2372                                                 bind, primary);
2373             bind;
2374             if (cdf == NULL)
2375                 return (0);
2376             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2377                                   cdf->cdf_type);
2378         }
2379     }
2380
2381     return (ret);
2382 }
2383
2384 static int
2385 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2386                           const char *file, const char *name, boolean_t primary, void *arg)
2387 {
2388     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2389                               const char *file, const char *name, void *arg)
2390     {
2391         int ret;
2392         uint_t bind, type;
2393         ctf_cu_t *cup = arg;
2394         int ret, bind, type;
2395
2396         bind = GELF_ST_BIND(symp->st_info);
2397         type = GELF_ST_TYPE(symp->st_info);
2398
2399         /*
2400          * Come back to weak symbols in another pass
2401          */
2402         if (bind == STB_WEAK)
2403             return (0);
2404
2405         if (type == STT_OBJECT) {
2406             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2407                                                 bind, primary);
2408             bind;
2409             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2410             if (cdv == NULL)
2411                 return (0);
2412             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2413             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2414             ctf_dprintf("added object %s\n", name);
2415         } else {
2416             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2417                                                 bind, primary);
2418             bind;
2419             if (cdf == NULL)
2420                 return (0);
2421             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2422                                   cdf->cdf_type);
2423         }
2424     }
2425
2426     return (ret);
2427 }
2428
2429 static int
2430 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2431                           const char *file, const char *name, boolean_t primary, void *arg)
2432 {
2433     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2434                               const char *file, const char *name, void *arg)
2435     {
2436         int ret;
2437         uint_t bind, type;
2438         ctf_cu_t *cup = arg;
2439         int ret, bind, type;
2440
2441         bind = GELF_ST_BIND(symp->st_info);
2442         type = GELF_ST_TYPE(symp->st_info);
2443
2444         /*
2445          * Come back to weak symbols in another pass
2446          */
2447         if (bind == STB_WEAK)
2448             return (0);
2449
2450         if (type == STT_OBJECT) {
2451             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2452                                                 bind, primary);
2453             bind;
2454             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2455             if (cdv == NULL)
2456                 return (0);
2457             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2458             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2459             ctf_dprintf("added object %s\n", name);
2460         } else {
2461             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2462                                                 bind, primary);
2463             bind;
2464             if (cdf == NULL)
2465                 return (0);
2466             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2467                                   cdf->cdf_type);
2468         }
2469     }
2470
2471     return (ret);
2472 }
2473
2474 static int
2475 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2476                           const char *file, const char *name, boolean_t primary, void *arg)
2477 {
2478     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2479                               const char *file, const char *name, void *arg)
2480     {
2481         int ret;
2482         uint_t bind, type;
2483         ctf_cu_t *cup = arg;
2484         int ret, bind, type;
2485
2486         bind = GELF_ST_BIND(symp->st_info);
2487         type = GELF_ST_TYPE(symp->st_info);
2488
2489         /*
2490          * Come back to weak symbols in another pass
2491          */
2492         if (bind == STB_WEAK)
2493             return (0);
2494
2495         if (type == STT_OBJECT) {
2496             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2497                                                 bind, primary);
2498             bind;
2499             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2500             if (cdv == NULL)
2501                 return (0);
2502             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2503             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2504             ctf_dprintf("added object %s\n", name);
2505         } else {
2506             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2507                                                 bind, primary);
2508             bind;
2509             if (cdf == NULL)
2510                 return (0);
2511             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2512                                   cdf->cdf_type);
2513         }
2514     }
2515
2516     return (ret);
2517 }
2518
2519 static int
2520 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2521                           const char *file, const char *name, boolean_t primary, void *arg)
2522 {
2523     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2524                               const char *file, const char *name, void *arg)
2525     {
2526         int ret;
2527         uint_t bind, type;
2528         ctf_cu_t *cup = arg;
2529         int ret, bind, type;
2530
2531         bind = GELF_ST_BIND(symp->st_info);
2532         type = GELF_ST_TYPE(symp->st_info);
2533
2534         /*
2535          * Come back to weak symbols in another pass
2536          */
2537         if (bind == STB_WEAK)
2538             return (0);
2539
2540         if (type == STT_OBJECT) {
2541             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2542                                                 bind, primary);
2543             bind;
2544             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2545             if (cdv == NULL)
2546                 return (0);
2547             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2548             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2549             ctf_dprintf("added object %s\n", name);
2550         } else {
2551             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2552                                                 bind, primary);
2553             bind;
2554             if (cdf == NULL)
2555                 return (0);
2556             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2557                                   cdf->cdf_type);
2558         }
2559     }
2560
2561     return (ret);
2562 }
2563
2564 static int
2565 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2566                           const char *file, const char *name, boolean_t primary, void *arg)
2567 {
2568     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2569                               const char *file, const char *name, void *arg)
2570     {
2571         int ret;
2572         uint_t bind, type;
2573         ctf_cu_t *cup = arg;
2574         int ret, bind, type;
2575
2576         bind = GELF_ST_BIND(symp->st_info);
2577         type = GELF_ST_TYPE(symp->st_info);
2578
2579         /*
2580          * Come back to weak symbols in another pass
2581          */
2582         if (bind == STB_WEAK)
2583             return (0);
2584
2585         if (type == STT_OBJECT) {
2586             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2587                                                 bind, primary);
2588             bind;
2589             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2590             if (cdv == NULL)
2591                 return (0);
2592             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2593             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2594             ctf_dprintf("added object %s\n", name);
2595         } else {
2596             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2597                                                 bind, primary);
2598             bind;
2599             if (cdf == NULL)
2600                 return (0);
2601             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2602                                   cdf->cdf_type);
2603         }
2604     }
2605
2606     return (ret);
2607 }
2608
2609 static int
2610 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2611                           const char *file, const char *name, boolean_t primary, void *arg)
2612 {
2613     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2614                               const char *file, const char *name, void *arg)
2615     {
2616         int ret;
2617         uint_t bind, type;
2618         ctf_cu_t *cup = arg;
2619         int ret, bind, type;
2620
2621         bind = GELF_ST_BIND(symp->st_info);
2622         type = GELF_ST_TYPE(symp->st_info);
2623
2624         /*
2625          * Come back to weak symbols in another pass
2626          */
2627         if (bind == STB_WEAK)
2628             return (0);
2629
2630         if (type == STT_OBJECT) {
2631             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2632                                                 bind, primary);
2633             bind;
2634             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2635             if (cdv == NULL)
2636                 return (0);
2637             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2638             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2639             ctf_dprintf("added object %s\n", name);
2640         } else {
2641             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2642                                                 bind, primary);
2643             bind;
2644             if (cdf == NULL)
2645                 return (0);
2646             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2647                                   cdf->cdf_type);
2648         }
2649     }
2650
2651     return (ret);
2652 }
2653
2654 static int
2655 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2656                           const char *file, const char *name, boolean_t primary, void *arg)
2657 {
2658     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2659                               const char *file, const char *name, void *arg)
2660     {
2661         int ret;
2662         uint_t bind, type;
2663         ctf_cu_t *cup = arg;
2664         int ret, bind, type;
2665
2666         bind = GELF_ST_BIND(symp->st_info);
2667         type = GELF_ST_TYPE(symp->st_info);
2668
2669         /*
2670          * Come back to weak symbols in another pass
2671          */
2672         if (bind == STB_WEAK)
2673             return (0);
2674
2675         if (type == STT_OBJECT) {
2676             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2677                                                 bind, primary);
2678             bind;
2679             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2680             if (cdv == NULL)
2681                 return (0);
2682             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2683             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2684             ctf_dprintf("added object %s\n", name);
2685         } else {
2686             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2687                                                 bind, primary);
2688             bind;
2689             if (cdf == NULL)
2690                 return (0);
2691             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2692                                   cdf->cdf_type);
2693         }
2694     }
2695
2696     return (ret);
2697 }
2698
2699 static int
2700 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2701                           const char *file, const char *name, boolean_t primary, void *arg)
2702 {
2703     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2704                               const char *file, const char *name, void *arg)
2705     {
2706         int ret;
2707         uint_t bind, type;
2708         ctf_cu_t *cup = arg;
2709         int ret, bind, type;
2710
2711         bind = GELF_ST_BIND(symp->st_info);
2712         type = GELF_ST_TYPE(symp->st_info);
2713
2714         /*
2715          * Come back to weak symbols in another pass
2716          */
2717         if (bind == STB_WEAK)
2718             return (0);
2719
2720         if (type == STT_OBJECT) {
2721             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2722                                                 bind, primary);
2723             bind;
2724             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2725             if (cdv == NULL)
2726                 return (0);
2727             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2728             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2729             ctf_dprintf("added object %s\n", name);
2730         } else {
2731             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2732                                                 bind, primary);
2733             bind;
2734             if (cdf == NULL)
2735                 return (0);
2736             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2737                                   cdf->cdf_type);
2738         }
2739     }
2740
2741     return (ret);
2742 }
2743
2744 static int
2745 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2746                           const char *file, const char *name, boolean_t primary, void *arg)
2747 {
2748     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2749                               const char *file, const char *name, void *arg)
2750     {
2751         int ret;
2752         uint_t bind, type;
2753         ctf_cu_t *cup = arg;
2754         int ret, bind, type;
2755
2756         bind = GELF_ST_BIND(symp->st_info);
2757         type = GELF_ST_TYPE(symp->st_info);
2758
2759         /*
2760          * Come back to weak symbols in another pass
2761          */
2762         if (bind == STB_WEAK)
2763             return (0);
2764
2765         if (type == STT_OBJECT) {
2766             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2767                                                 bind, primary);
2768             bind;
2769             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2770             if (cdv == NULL)
2771                 return (0);
2772             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2773             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2774             ctf_dprintf("added object %s\n", name);
2775         } else {
2776             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2777                                                 bind, primary);
2778             bind;
2779             if (cdf == NULL)
2780                 return (0);
2781             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2782                                   cdf->cdf_type);
2783         }
2784     }
2785
2786     return (ret);
2787 }
2788
2789 static int
2790 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2791                           const char *file, const char *name, boolean_t primary, void *arg)
2792 {
2793     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2794                               const char *file, const char *name, void *arg)
2795     {
2796         int ret;
2797         uint_t bind, type;
2798         ctf_cu_t *cup = arg;
2799         int ret, bind, type;
2800
2801         bind = GELF_ST_BIND(symp->st_info);
2802         type = GELF_ST_TYPE(symp->st_info);
2803
2804         /*
2805          * Come back to weak symbols in another pass
2806          */
2807         if (bind == STB_WEAK)
2808             return (0);
2809
2810         if (type == STT_OBJECT) {
2811             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2812                                                 bind, primary);
2813             bind;
2814             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2815             if (cdv == NULL)
2816                 return (0);
2817             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2818             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2819             ctf_dprintf("added object %s\n", name);
2820         } else {
2821             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2822                                                 bind, primary);
2823             bind;
2824             if (cdf == NULL)
2825                 return (0);
2826             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2827                                   cdf->cdf_type);
2828         }
2829     }
2830
2831     return (ret);
2832 }
2833
2834 static int
2835 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2836                           const char *file, const char *name, boolean_t primary, void *arg)
2837 {
2838     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2839                               const char *file, const char *name, void *arg)
2840     {
2841         int ret;
2842         uint_t bind, type;
2843         ctf_cu_t *cup = arg;
2844         int ret, bind, type;
2845
2846         bind = GELF_ST_BIND(symp->st_info);
2847         type = GELF_ST_TYPE(symp->st_info);
2848
2849         /*
2850          * Come back to weak symbols in another pass
2851          */
2852         if (bind == STB_WEAK)
2853             return (0);
2854
2855         if (type == STT_OBJECT) {
2856             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2857                                                 bind, primary);
2858             bind;
2859             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2860             if (cdv == NULL)
2861                 return (0);
2862             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2863             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2864             ctf_dprintf("added object %s\n", name);
2865         } else {
2866             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2867                                                 bind, primary);
2868             bind;
2869             if (cdf == NULL)
2870                 return (0);
2871             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2872                                   cdf->cdf_type);
2873         }
2874     }
2875
2876     return (ret);
2877 }
2878
2879 static int
2880 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2881                           const char *file, const char *name, boolean_t primary, void *arg)
2882 {
2883     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2884                               const char *file, const char *name, void *arg)
2885     {
2886         int ret;
2887         uint_t bind, type;
2888         ctf_cu_t *cup = arg;
2889         int ret, bind, type;
2890
2891         bind = GELF_ST_BIND(symp->st_info);
2892         type = GELF_ST_TYPE(symp->st_info);
2893
2894         /*
2895          * Come back to weak symbols in another pass
2896          */
2897         if (bind == STB_WEAK)
2898             return (0);
2899
2900         if (type == STT_OBJECT) {
2901             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2902                                                 bind, primary);
2903             bind;
2904             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2905             if (cdv == NULL)
2906                 return (0);
2907             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2908             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2909             ctf_dprintf("added object %s\n", name);
2910         } else {
2911             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2912                                                 bind, primary);
2913             bind;
2914             if (cdf == NULL)
2915                 return (0);
2916             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2917                                   cdf->cdf_type);
2918         }
2919     }
2920
2921     return (ret);
2922 }
2923
2924 static int
2925 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2926                           const char *file, const char *name, boolean_t primary, void *arg)
2927 {
2928     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2929                               const char *file, const char *name, void *arg)
2930     {
2931         int ret;
2932         uint_t bind, type;
2933         ctf_cu_t *cup = arg;
2934         int ret, bind, type;
2935
2936         bind = GELF_ST_BIND(symp->st_info);
2937         type = GELF_ST_TYPE(symp->st_info);
2938
2939         /*
2940          * Come back to weak symbols in another pass
2941          */
2942         if (bind == STB_WEAK)
2943             return (0);
2944
2945         if (type == STT_OBJECT) {
2946             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2947                                                 bind, primary);
2948             bind;
2949             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2950             if (cdv == NULL)
2951                 return (0);
2952             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2953             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2954             ctf_dprintf("added object %s\n", name);
2955         } else {
2956             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
2957                                                 bind, primary);
2958             bind;
2959             if (cdf == NULL)
2960                 return (0);
2961             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
2962                                   cdf->cdf_type);
2963         }
2964     }
2965
2966     return (ret);
2967 }
2968
2969 static int
2970 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
2971                           const char *file, const char *name, boolean_t primary, void *arg)
2972 {
2973     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
2974                               const char *file, const char *name, void *arg)
2975     {
2976         int ret;
2977         uint_t bind, type;
2978         ctf_cu_t *cup = arg;
2979         int ret, bind, type;
2980
2981         bind = GELF_ST_BIND(symp->st_info);
2982         type = GELF_ST_TYPE(symp->st_info);
2983
2984         /*
2985          * Come back to weak symbols in another pass
2986          */
2987         if (bind == STB_WEAK)
2988             return (0);
2989
2990         if (type == STT_OBJECT) {
2991             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
2992                                                 bind, primary);
2993             bind;
2994             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
2995             if (cdv == NULL)
2996                 return (0);
2997             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
2998             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
2999             ctf_dprintf("added object %s\n", name);
3000         } else {
3001             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
3002                                                 bind, primary);
3003             bind;
3004             if (cdf == NULL)
3005                 return (0);
3006             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
3007                                   cdf->cdf_type);
3008         }
3009     }
3010
3011     return (ret);
3012 }
3013
3014 static int
3015 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
3016                           const char *file, const char *name, boolean_t primary, void *arg)
3017 {
3018     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
3019                               const char *file, const char *name, void *arg)
3020     {
3021         int ret;
3022         uint_t bind, type;
3023         ctf_cu_t *cup = arg;
3024         int ret, bind, type;
3025
3026         bind = GELF_ST_BIND(symp->st_info);
3027         type = GELF_ST_TYPE(symp->st_info);
3028
3029         /*
3030          * Come back to weak symbols in another pass
3031          */
3032         if (bind == STB_WEAK)
3033             return (0);
3034
3035         if (type == STT_OBJECT) {
3036             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
3037                                                 bind, primary);
3038             bind;
3039             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
3040             if (cdv == NULL)
3041                 return (0);
3042             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
3043             ctf_dprintf("added object %s->%ld\n", name, cdv->cdv_type);
3044             ctf_dprintf("added object %s\n", name);
3045         } else {
3046             ctf_dfunc_t *cdf = ctf_dwarf_match_func(cup, file, name,
3047                                                 bind, primary);
3048             bind;
3049             if (cdf == NULL)
3050                 return (0);
3051             ret = ctf_add_function(cup->cu_ctfp, idx, &cdf->cdf_fip,
3052                                   cdf->cdf_type);
3053         }
3054     }
3055
3056     return (ret);
3057 }
3058
3059 static int
3060 ctf_dwarf_conv_funcvars_cb(const Elf64_Sym *symp, ulong_t idx,
3061                           const char *file, const char *name, boolean_t primary, void *arg)
3062 {
3063     ctf_dwarf_conv_funcvars_cb(ctf_cu_t *cup, const GElf_Sym *symp, ulong_t idx,
3064                               const char *file, const char *name, void *arg)
3065     {
3066         int ret;
3067         uint_t bind, type;
3068         ctf_cu_t *cup = arg;
3069         int ret, bind, type;
3070
3071         bind = GELF_ST_BIND(symp->st_info);
3072         type = GELF_ST_TYPE(symp->st_info);
3073
3074         /*
3075          * Come back to weak symbols in another pass
3076          */
3077         if (bind == STB_WEAK)
3078             return (0);
3079
3080         if (type == STT_OBJECT) {
3081             ctf_dwarf_t *cdv = ctf_dwarf_match_var(cup, file, name,
3082                                                 bind, primary);
3083             bind;
3084             ctf_dprintf("match for %s (%d): %p\n", name, idx, cdv);
3085             if (cdv == NULL)
3086                 return (0);
3087             ret = ctf_add_object(cup->cu_ctfp, idx, cdv->cdv_type);
3088             c
```

```

2315         cdf->cdf_argv);
2316         ctf_dprintf("added function %s\n", name);
2317     }
2318
2319     if (ret == CTF_ERR) {
2320         return (ctf_errno(cup->cu_ctfp));
2321     }
2322
2323     return (0);
2324 }
2325
2326 static int
2327 ctf_dwarf_conv_funcvars(ctf_cu_t *cup)
2328 {
2329     return (ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_funcvars_cb, cup));
2330     return (ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_funcvars_cb, NULL));
2331 }
2332 /*
2333 * If we have a weak symbol, attempt to find the strong symbol it will resolve
2334 * to. Note: the code where this actually happens is in sym_process() in
2335 * cmd/sgs/libld/common/syms.c
2336 *
2337 * Finding the matching symbol is unfortunately not trivial. For a symbol to be
2338 * a candidate, it must:
2339 *
2340 * - have the same type (function, object)
2341 * - have the same value (address)
2342 * - have the same size
2343 * - not be another weak symbol
2344 * - belong to the same section (checked via section index)
2345 *
2346 * To perform this check, we first iterate over the symbol table. For each weak
2347 * symbol that we encounter, we then do a second walk over the symbol table,
2348 * calling ctf_dwarf_conv_check_weak(). If a symbol matches the above, then it's
2349 * either a local or global symbol. If we find a global symbol then we go with
2350 * it and stop searching for additional matches.
2351 *
2352 * If instead, we find a local symbol, things are more complicated. The first
2353 * thing we do is to try and see if we have file information about both symbols
2354 * (STT_FILE). If they both have file information and it matches, then we treat
2355 * that as a good match and stop searching for additional matches.
2356 *
2357 * Otherwise, this means we have a non-matching file and a local symbol. We
2358 * treat this as a candidate and if we find a better match (one of the two cases
2359 * above), use that instead. There are two different ways this can happen.
2360 * Either this is a completely different symbol, or it's a once-global symbol
2361 * that was scoped to local via a mapfile. In the former case, curfile is
2362 * likely inaccurate since the linker does not preserve the needed curfile in
2363 * the order of the symbol table (see the comments about locally scoped symbols
2364 * in libld's update_osym()). As we can't tell this case from the former one,
2365 * we use this symbol iff no other matching symbol is found.
2366 *
2367 * What we really need here is a SUNW section containing weak<->strong mappings
2368 * that we can consume.
2369 */
2370 typedef struct ctf_dwarf_weak_arg {
2371     const Elf64_Sym *cweak_symp;
2372     const GElf_Sym *cweak_symp;
2373     const char *cweak_file;
2374     boolean_t cweak_candidate;
2375     ulong_t cweak_idx;
2376 } ctf_dwarf_weak_arg_t;
2377
2378 static int
2379 ctf_dwarf_conv_check_weak(const Elf64_Sym *symp, ulong_t idx, const char *file,

```

```

2379     const char *name, boolean_t primary, void *arg)
2380     ctf_dwarf_conv_check_weak(ctf_cu_t *cup, const GElf_Sym *symp,
2381     ulong_t idx, const char *file, const char *name, void *arg)
2382 {
2383     ctf_dwarf_weak_arg_t *cweak = arg;
2384     const GElf_Sym *wsymp = cweak->cweak_symp;
2385
2386     const Elf64_Sym *wsymp = cweak->cweak_symp;
2387
2388     ctf_dprintf("comparing weak to %s\n", name);
2389
2390     if (GELF_ST_BIND(symp->st_info) == STB_WEAK) {
2391         return (0);
2392     }
2393
2394     if (GELF_ST_TYPE(wsymp->st_info) != GELF_ST_TYPE(symp->st_info)) {
2395         return (0);
2396     }
2397
2398     if (wsymp->st_value != symp->st_value) {
2399         return (0);
2400     }
2401
2402     if (wsymp->st_size != symp->st_size) {
2403         return (0);
2404     }
2405
2406     /*
2407      * Check if it's a weak candidate.
2408      */
2409     if (GELF_ST_BIND(symp->st_info) == STB_LOCAL &&
2410     (file == NULL || cweak->cweak_file == NULL ||
2411     strcmp(file, cweak->cweak_file) != 0)) {
2412         cweak->cweak_candidate = B_TRUE;
2413         cweak->cweak_idx = idx;
2414         return (0);
2415     }
2416
2417     /*
2418      * Found a match, break.
2419      */
2420     cweak->cweak_idx = idx;
2421
2422     return (1);
2423 }
2424
2425 unchanged_portion_omitted
2426
2427 static int
2428 ctf_dwarf_conv_weaks_cb(const Elf64_Sym *symp, ulong_t idx, const char *file,
2429     const char *name, boolean_t primary, void *arg)
2430     ctf_dwarf_conv_weaks_cb(ctf_cu_t *cup, const GElf_Sym *symp,
2431     ulong_t idx, const char *file, const char *name, void *arg)
2432 {
2433     int ret, type;
2434     ctf_dwarf_weak_arg_t cweak;
2435     ctf_cu_t *cup = arg;
2436
2437     /*
2438      * We only care about weak symbols.
2439      */
2440     if (GELF_ST_BIND(symp->st_info) != STB_WEAK)
2441         return (0);

```

new/usr/src/lib/libctf/common/ctf\_dwarf.c

11

```
2496     type = GELF_ST_TYPE(symp->st_info);
2497     ASSERT(type == STT_OBJECT || type == STT_FUNC);
2498
2499     /*
2500      * For each weak symbol we encounter, we need to do a second iteration
2501      * to try and find a match. We should probably think about other
2502      * techniques to try and save us time in the future.
2503      */
2504     cweak.cweak_symp = symp;
2505     cweak.cweak_file = file;
2506     cweak.cweak_candidate = B_FALSE;
2507     cweak.cweak_idx = 0;
2508
2509     ctf_dprintf("Trying to find weak equiv for %s\n", name);
2510
2511     ret = ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_check_weak, &cweak);
2512     ret = ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_check_weak, &cweak);
2513     VERIFY(ret == 0 || ret == 1);
2514
2515     /*
2516      * Nothing was ever found, we're not going to add anything for this
2517      * entry.
2518      */
2519     if (ret == 0 && cweak.cweak_candidate == B_FALSE) {
2520         ctf_dprintf("found no weak match for %s\n", name);
2521         return (0);
2522     }
2523
2524     /*
2525      * Now, finally go and add the type based on the match.
2526      */
2527     ctf_dprintf("matched weak symbol %lu to %lu\n", idx, cweak.cweak_idx);
2528     if (type == STT_OBJECT) {
2529         ret = ctf_dwarf_duplicate_sym(cup, idx, cweak.cweak_idx);
2530     } else {
2531         ret = ctf_dwarf_duplicate_func(cup, idx, cweak.cweak_idx);
2532     }
2533
2534     return (ret);
2535 }
2536 static int
2537 ctf_dwarf_conv_weaks(ctf_cu_t *cup)
2538 {
2539     return (ctf_symtab_iter(cup->cu_ctfp, ctf_dwarf_conv_weaks_cb, cup));
2540     return (ctf_dwarf_symtab_iter(cup, ctf_dwarf_conv_weaks_cb, NULL));
2541 }
2542 /* ARGSUSED */
2543 static int
2544 ctf_dwarf_convert_one(void *arg, void *unused)
2545 {
2546     int ret;
2547     ctf_file_t *dedup;
2548     ctf_cu_t *cup = arg;
2549
2550     ctf_dprintf("converting die: %s\n", cup->cu_name);
2551     ctf_dprintf("max offset: %x\n", cup->cu_maxoff);
2552     VERIFY(cup != NULL);
2553
2554     ret = ctf_dwarf_convert_die(cup, cup->cu_cu);
2555     ctf_dprintf("ctf_dwarf_convert_die (%s) returned %d\n", cup->cu_name,
2556                 ret);
2557     if (ret != 0) {
2558         return (ret);
2559     }
```

new/usr/src/lib/libctf/common/ctf\_dwarf.c

12

```
2560     if (ctf_update(cup->cu_ctfp) != 0) {
2561         return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2562                                "failed to update output ctf container"));
2563     }
2564
2565     ret = ctf_dwarf_fixup_die(cup, B_FALSE);
2566     ctf_dprintf("ctf_dwarf_fixup_die (%s) returned %d\n", cup->cu_name,
2567                 ret);
2568     if (ret != 0) {
2569         return (ret);
2570     }
2571     if (ctf_update(cup->cu_ctfp) != 0) {
2572         return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2573                                "failed to update output ctf container"));
2574     }
2575
2576     ret = ctf_dwarf_fixup_die(cup, B_TRUE);
2577     ctf_dprintf("ctf_dwarf_fixup_die (%s) returned %d\n", cup->cu_name,
2578                 ret);
2579     if (ret != 0) {
2580         return (ret);
2581     }
2582     if (ctf_update(cup->cu_ctfp) != 0) {
2583         return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2584                                "failed to update output ctf container"));
2585     }
2586
2587     if ((ret = ctf_dwarf_conv_funcvars(cup)) != 0) {
2588         return (ctf_dwarf_error(cup, NULL, ret,
2589                                "failed to convert strong functions and variables"));
2590     }
2591
2592     if (ctf_update(cup->cu_ctfp) != 0) {
2593         return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2594                                "failed to update output ctf container"));
2595     }
2596
2597     if (cup->cu_doweaks == B_TRUE) {
2598         if ((ret = ctf_dwarf_conv_weaks(cup)) != 0) {
2599             return (ctf_dwarf_error(cup, NULL, ret,
2600                                    "failed to convert weak functions and variables"));
2601         }
2602
2603         if (ctf_update(cup->cu_ctfp) != 0) {
2604             return (ctf_dwarf_error(cup, cup->cu_ctfp, 0,
2605                                    "failed to update output ctf container"));
2606         }
2607     }
2608
2609     ctf_phase_dump(cup->cu_ctfp, "pre-dwarf-dedup", cup->cu_name);
2610     ctf_phase_dump(cup->cu_ctfp, "pre-dedup");
2611     ctf_dprintf("adding inputs for dedup\n");
2612     if ((ret = ctf_merge_add(cup->cu_cmh, cup->cu_ctfp)) != 0) {
2613         return (ctf_dwarf_error(cup, NULL, ret,
2614                                "failed to add inputs for merge"));
2615     }
2616
2617     ctf_dprintf("starting dedup of %s\n", cup->cu_name);
2618     ctf_dprintf("starting merge\n");
2619     if ((ret = ctf_merge_dedup(cup->cu_cmh, &dedup)) != 0) {
2620         return (ctf_dwarf_error(cup, NULL, ret,
2621                                "failed to deduplicate die"));
2622     }
2623     ctf_close(cup->cu_ctfp);
2624     cup->cu_ctfp = dedup;
```

```

2624     ctf_phase_dump(cup->cu_ctfp, "post-dwarf-dedup", cup->cu_name);
2626     return (0);
2627 }
unchanged_portion_omitted_

2839 ctf_conv_status_t
2840 ctf_dwarf_convert(int fd, Elf *elf, uint_t nthrs, int *errp, ctf_file_t **fpp,
2841     char *errmsg, size_t errlen)
2842 {
2843     int err, ret, ndies, i;
2844     Dwarf_Debug dw;
2845     Dwarf_Error derr;
2846     ctf_cu_t *cdies = NULL, *cup;
2847     workq_t *wqp = NULL;
2848
2849     if (errp == NULL)
2850         errp = &err;
2851     *errp = 0;
2852     *fpp = NULL;
2853
2854     ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL, &dw, &derr);
2855     if (ret != DW_DLV_OK) {
2856         /*
2857          * We may want to expect DWARF data here and fail conversion if
2858          * it's missing. In this case, if we actually have some amount
2859          * of DWARF, but no section, for now, just go ahead and create
2860          * an empty CTF file.
2861         */
2862     if (ret == DW_DLV_NO_ENTRY ||
2863         dwarf_errno(derr) == DW_DLE_DEBUG_INFO_NULL) {
2864         *fpp = ctf_create(errp);
2865         return (*fpp != NULL ? CTF_CONV_SUCCESS :
2866             CTF_CONV_ERROR);
2867     }
2868     (void) sprintf(errmsg, errlen,
2869         "failed to initialize DWARF: %s\n",
2870         dwarf_errmsg(derr));
2871     *errp = ECTF_CONVBKERR;
2872     return (CTF_CONV_ERROR);
2873 }
2874
2875 /*
2876  * Iterate over all of the compilation units and create a ctf_cu_t for
2877  * each of them. This is used to determine if we have zero, one, or
2878  * multiple dies to convert. If we have zero, that's an error. If
2879  * there's only one die, that's the simple case. No merge needed and
2880  * only a single Dwarf_Debug as well.
2881 */
2882 ndies = 0;
2883 ret = ctf_dwarf_count_dies(dw, &derr, &ndies, errmsg, errlen);
2884 if (ret != 0) {
2885     *errp = ret;
2886     goto out;
2887 }
2888
2889 (void) dwarf_finish(dw, &derr);
2890 cdies = ctf_alloc(sizeof(ctf_cu_t) * ndies);
2891 if (cdies == NULL) {
2892     *errp = ENOMEM;
2893     return (CTF_CONV_ERROR);
2894 }
2895
2896 for (i = 0; i < ndies; i++) {
2897     cup = &cdies[i];

```

```

2898     ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL,
2899         &cup->cu_dwarf, &derr);
2900     if (ret != 0) {
2901         ctf_free(cdies, sizeof(ctf_cu_t) * ndies);
2902         (void) sprintf(errmsg, errlen,
2903             "failed to initialize DWARF: %s\n",
2904             dwarf_errmsg(derr));
2905         *errp = ECTF_CONVBKERR;
2906         return (CTF_CONV_ERROR);
2907     }
2908
2909     ret = ctf_dwarf_init_die(fd, elf, &cdies[i], i, errmsg, errlen);
2910     if (ret != 0) {
2911         *errp = ret;
2912         goto out;
2913     }
2914
2915     cup->cu_doweaks = ndies > 1 ? B_FALSE : B_TRUE;
2916 }
2917
2918 ctf_dprintf("found %d DWARF CU's\n", ndies);
2919 ctf_dprintf("found %d DWARF die(s)\n", ndies);
2920
2921 /*
2922  * If we only have one compilation unit, there's no reason to use
2923  * multiple threads, even if the user requested them. After all, they
2924  * just gave us an upper bound.
2925 */
2926 if (ndies == 1)
2927     nthrs = 1;
2928
2929 if (workq_init(&wqp, nthrs) == -1) {
2930     *errp = errno;
2931     goto out;
2932 }
2933
2934 for (i = 0; i < ndies; i++) {
2935     cup = &cdies[i];
2936     ctf_dprintf("adding cu %s: %p, %x %x\n", cup->cu_name,
2937     ctf_dprintf("adding die %s: %p, %x %x\n", cup->cu_name,
2938         cup->cu_cu, cup->cu_cuoff, cup->cu_maxoff);
2939     if (workq_add(wqp, cup) == -1) {
2940         *errp = errno;
2941         goto out;
2942     }
2943
2944     ret = workq_work(wqp, ctf_dwarf_convert_one, NULL, errp);
2945     if (ret == WORKQ_ERROR) {
2946         *errp = errno;
2947         goto out;
2948     } else if (ret == WORKQ_UERROR) {
2949         ctf_dprintf("internal convert failed: %s\n",
2950             ctf_errmsg(*errp));
2951         goto out;
2952     }
2953
2954 ctf_dprintf("Determining next phase: have %d CU's\n", ndies);
2955 ctf_dprintf("Determining next phase: have %d dies\n", ndies);
2956 if (ndies != 1) {
2957     ctf_merge_t *cmp;
2958
2959     cmp = ctf_merge_init(fd, &ret);
2960     if (cmp == NULL) {
2961         *errp = ret;
2962         goto out;
2963     }

```

```
2961         }
2962         ctf_dprintf("setting threads\n");
2963         if ((ret = ctf_merge_set_nthreads(cmp, nthrs)) != 0) {
2964             ctf_merge_fini(cmp);
2965             *errp = ret;
2966             goto out;
2967         }
2968     }
2969     ctf_dprintf("adding dies\n");
2970     for (i = 0; i < ndies; i++) {
2971         cup = &cdies[i];
2972         ctf_dprintf("adding cu %s (%p)\n", cup->cu_name,
2973                     cup->cu_ctfp);
2974         if ((ret = ctf_merge_add(cmp, cup->cu_ctfp)) != 0) {
2975             ctf_merge_fini(cmp);
2976             *errp = ret;
2977             goto out;
2978         }
2979     }
2980     ctf_dprintf("performing merge\n");
2981     ret = ctf_merge_merge(cmp, fpp);
2982     if (ret != 0) {
2983         ctf_dprintf("failed merge!\n");
2984         *fpp = NULL;
2985         ctf_merge_fini(cmp);
2986         *errp = ret;
2987         goto out;
2988     }
2989     ctf_merge_fini(cmp);
2990     *errp = 0;
2991     ctf_dprintf("successfully converted!\n");
2992 } else {
2993     *errp = 0;
2994     *fpp = cdies->cu_ctfp;
2995     cdies->cu_ctfp = NULL;
2996     ctf_dprintf("successfully converted!\n");
2997 }
2998 }
```

unchanged portion omitted

new/usr/src/lib/libctf/common/ctf\_lib.c

```
*****  
21605 Tue Apr 23 05:24:01 2019  
new/usr/src/lib/libctf/common/ctf_lib.c  
10812 ctf tools shouldn't add blank labels  
10813 ctf symbol mapping needs work  
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>  
*****  
1 /*  
2  * CDDL HEADER START  
3  *  
4  * The contents of this file are subject to the terms of the  
5  * Common Development and Distribution License, Version 1.0 only  
6  * (the "License"). You may not use this file except in compliance  
7  * with the License.  
8  *  
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE  
10 * or http://www.opensolaris.org/os/licensing.  
11 * See the License for the specific language governing permissions  
12 * and limitations under the License.  
13 *  
14 * When distributing Covered Code, include this CDDL HEADER in each  
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.  
16 * If applicable, add the following below this CDDL HEADER, with the  
17 * fields enclosed by brackets "[]" replaced with your own identifying  
18 * information: Portions Copyright [yyyy] [name of copyright owner]  
19 *  
20 * CDDL HEADER END  
21 */  
22 /*  
23 * Copyright 2003 Sun Microsystems, Inc. All rights reserved.  
24 * Use is subject to license terms.  
25 */  
26 /*  
27 * Copyright (c) 2019, Joyent, Inc.  
27 * Copyright (c) 2015, Joyent, Inc.  
28 */  
  
30 #include <sys/types.h>  
31 #include <sys/stat.h>  
32 #include <sys/mman.h>  
33 #include <libctf_impl.h>  
33 #include <ctf_impl.h>  
34 #include <unistd.h>  
35 #include <fcntl.h>  
36 #include <errno.h>  
37 #include <dlopen.h>  
38 #include <elf.h>  
39 #include <zlib.h>  
40 #include <sys/debug.h>  
  
42 #ifdef _LP64  
43 static const char *_libctf_zlib = "/usr/lib/64/libz.so.1";  
44 #else  
45 static const char *_libctf_zlib = "/usr/lib/libz.so.1";  
46 #endif  
  
48 static struct {  
49     int (*z_uncompress)(uchar_t *, ulong_t *, const uchar_t *, ulong_t);  
50     int (*z_initcomp)(z_stream *, int, const char *, int);  
51     int (*z_compress)(z_stream *, int);  
52     int (*z_finicomp)(z_stream *);  
53     const char *(*z_error)(int);  
54     void *z_dlp;  
55 } zlib;  
unchanged_portion_omitted_
```

1

```
new/usr/src/lib/libctf/common/ctf_lib.c  
*****  
746 /*  
747  * A utility function for folks debugging CTF conversion and merging.  
748  */  
749 void  
750 ctf_phase_dump(ctf_file_t *fp, const char *phase, const char *name)  
750 ctf_phase_dump(ctf_file_t *fp, const char *phase)  
751 {  
752     int fd;  
753     static char *base;  
754     char path[MAXPATHLEN];  
756     if (base == NULL && (base = getenv("LIBCTF_WRITE_PHASES")) == NULL)  
757         return;  
759     if (name == NULL)  
760         name = "libctf";  
762     (void) sprintf(path, sizeof(path), "%s/%s.%s.%d.ctf", base, name,  
759     (void) sprintf(path, sizeof(path), "%s/libctf.%s.%d.ctf", base,  
763         phase != NULL ? phase : "",  
764         ctf_phase);  
765     if ((fd = open(path, O_CREAT | O_TRUNC | O_RDWR, 0777)) < 0)  
766         return;  
767     (void) ctf_write(fp, fd);  
768     (void) close(fd);  
769 }  
771 void  
772 ctf_phase_bump(void)  
773 {  
774     ctf_phase++;  
775 }  
777 int  
778 ctf_symtab_iter(ctf_file_t *fp, ctf_symtab_f func, void *arg)  
779 {  
780     ulong_t i;  
781     uintptr_t symbase;  
782     uintptr_t strbase;  
783     const char *file = NULL;  
784     boolean_t primary = B_TRUE;  
786     if (fp->ctf_symtab.cts_data == NULL ||  
787         fp->ctf_strtab.cts_data == NULL) {  
788         return (ECTF_NOSYMTAB);  
789     }  
791     symbase = (uintptr_t)fp->ctf_symtab.cts_data;  
792     strbase = (uintptr_t)fp->ctf_strtab.cts_data;  
794     for (i = 0; i < fp->ctf_nsyms; i++) {  
795         const char *name;  
796         int ret;  
797         uint_t type;  
798         Elf64_Sym sym;  
800         /*  
801          * The CTF library has historically tried to handle large file  
802          * offsets itself so that way clients can be unaware of such  
803          * issues. Therefore, we translate everything to a 64-bit ELF  
804          * symbol, this is done to make it so that the rest of the  
805          * library doesn't have to know about these differences. For  
806          * more information see, lib/libctf/common/ctf_lib.c.  
807          */  
808     if (fp->ctf_symtab.cts_entsize == sizeof(Elf32_Sym)) {  
809         const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
```

2

```
810         uint_t bind, itype;
811
812         sym.st_name = symp->st_name;
813         sym.st_value = symp->st_value;
814         sym.st_size = symp->st_size;
815         bind = ELF32_ST_BIND(symp->st_info);
816         itype = ELF32_ST_TYPE(symp->st_info);
817         sym.st_info = ELF64_ST_INFO(bind, itype);
818         sym.st_other = symp->st_other;
819         sym.st_shndx = symp->st_shndx;
820     } else {
821         const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;
822
823         sym = *symp;
824     }
825
826     type = ELF64_ST_TYPE(sym.st_info);
827     name = (const char *)(strbase + sym.st_name);
828
829     /*
830      * Check first if we have an STT_FILE entry. This is used to
831      * distinguish between various local symbols when merging.
832      */
833     if (type == STT_FILE) {
834         if (file != NULL) {
835             primary = B_FALSE;
836         }
837         file = name;
838         continue;
839     }
840
841     /*
842      * Check if this is a symbol that we care about.
843      */
844     if (!ctf_sym_valid(strbase, type, sym.st_shndx, sym.st_value,
845                        sym.st_name)) {
846         continue;
847     }
848
849     if ((ret = func(&sym, i, file, name, primary, arg)) != 0) {
850         return (ret);
851     }
852 }
853
854 return (0);
855 }
```

unchanged\_portion\_omitted\_

new/usr/src/lib/libctf/common/ctf\_merge.c

```
*****  
43216 Tue Apr 23 05:24:02 2019  
new/usr/src/lib/libctf/common/ctf_merge.c  
10812 ctf tools shouldn't add blank labels  
10813 ctf symbol mapping needs work  
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>  
*****  
1 /*  
2  * This file and its contents are supplied under the terms of the  
3  * Common Development and Distribution License (" CDDL"), version 1.0.  
4  * You may only use this file in accordance with the terms of version  
5  * 1.0 of the CDDL.  
6  *  
7  * A full copy of the text of the CDDL should have accompanied this  
8  * source. A copy of the CDDL is also available via the Internet at  
9  * http://www.illumos.org/license/CDDL.  
10 */  
  
12 /*  
13  * Copyright (c) 2019 Joyent, Inc.  
13  * Copyright (c) 2015 Joyent, Inc.  
14 */  
  
16 /*  
17  * To perform a merge of two CTF containers, we first diff the two containers  
18  * types. For every type that's in the src container, but not in the dst  
19  * container, we note it and add it to dst container. If there are any objects  
20  * or functions associated with src, we go through and update the types that  
21  * they refer to such that they all refer to types in the dst container.  
22 *  
23 * The bulk of the logic for the merge, after we've run the diff, occurs in  
24 * ctf_merge_common().  
25 *  
26 * In terms of exported APIs, we don't really export a simple merge two  
27 * containers, as the general way this is used, in something like ctfmerge(1),  
28 * is to add all the containers and then let us figure out the best way to merge  
29 * it.  
30 */  
  
32 #include <libctf_impl.h>  
33 #include <sys/debug.h>  
34 #include <sys/list.h>  
35 #include <stddef.h>  
36 #include <fcntl.h>  
37 #include <sys/types.h>  
38 #include <sys/stat.h>  
39 #include <mergeq.h>  
40 #include <errno.h>  
  
42 typedef struct ctf_merge_tinfo {  
43     uint16_t cmt_map;           /* Map to the type in out */  
44     boolean_t cmt_fixup;  
45     boolean_t cmt_forward;  
46     boolean_t cmt_missing;  
47 } ctf_merge_tinfo_t;  
unchanged_portion_omitted  
  
60 typedef struct ctf_merge_objmap {  
61     list_node_t cmo_node;  
62     const char *cmo_name;      /* Symbol name */  
63     const char *cmo_file;      /* Symbol file */  
64     ulong_t cmo_idx;          /* Symbol ID */  
65     Elf64_Sym cmo_sym;        /* Symbol Entry */  
66     ctf_id_t cmo_tid;         /* Type ID */  
67 } ctf_merge_objmap_t;
```

new/usr/src/lib/libctf/common/ctf\_merge.c

```

69 typedef struct ctf_merge_funcmap {
70     list_node_t cmf_node;
71     const char *cmf_name; /* Symbol name */
72     const char *cmf_file; /* Symbol file */
73     ulong_t cmf_idx; /* Symbol ID */
74     Elf64_Sym cmf_sym; /* Symbol Entry */
75     ctf_id_t cmf_rtid; /* Type ID */
76     uint_t cmf_flags; /* ctf_funcinfo_t ctc_flags */
77     uint_t cmf_argc; /* Number of arguments */
78     ctf_id_t cmf_args[]; /* Types of arguments */
79 } ctf_merge_funcmap_t;
unchanged_portion_omitted

101 typedef struct ctf_merge_symbol_arg {
102     list_t *cmsa_objmap;
103     list_t *cmsa_funcmap;
104     ctf_file_t *cmsa_out;
105     boolean_t cmsa_dedup;
106 } ctf_merge_symbol_arg_t;

108 static int ctf_merge_add_type(ctf_merge_types_t *, ctf_id_t);

110 static ctf_id_t
111 ctf_merge_gettype(ctf_merge_types_t *cmp, ctf_id_t id)
112 {
113     if (cmp->cm_dedup == B_FALSE) {
114         VERIFY(cmp->cm_tmap[id].cmt_map != 0);
115         return (cmp->cm_tmap[id].cmt_map);
116     }

117     while (cmp->cm_tmap[id].cmt_missing == B_FALSE) {
118         VERIFY(cmp->cm_tmap[id].cmt_map != 0);
119         id = cmp->cm_tmap[id].cmt_map;
120     }
121     VERIFY(cmp->cm_tmap[id].cmt_map != 0);
122     return (cmp->cm_tmap[id].cmt_map);
123 }
unchanged_portion_omitted

653 /*
654 * We're going to do three passes over the containers.
655 *
656 * Pass 1 checks for forward references in the output container that we know
657 * exist in the source container.
658 *
659 * Pass 2 adds all the missing types from the source container. As part of this
660 * we may be adding a type as a forward reference that doesn't exist yet.
661 * Any types that we encounter in this form, we need to add to a third pass.
662 *
663 * Pass 3 is the fixup pass. Here we go through and find all the types that were
664 * missing in the first.
665 *
666 * Importantly, we *must* call ctf_update between the second and third pass,
667 * otherwise several of the libctf functions will not properly find the data in
668 * the container. If we're doing a dedup we also fix up the type mapping.
669 */
670 static int
671 ctf_merge_common(ctf_merge_types_t *cmp)
672 {
673     int ret, i;

675     ctf_phase_dump(cmp->cm_src, "merge-common-src", NULL);
676     ctf_phase_dump(cmp->cm_out, "merge-common-dest", NULL);
677     ctf_phase_dump(cmp->cm_src, "merge-common-src");
678     ctf_phase_dump(cmp->cm_out, "merge-common-dest");

```

```

678     /* Pass 1 */
679     for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
680         if (cmp->cm_tmap[i].cmt_forward == B_TRUE) {
681             ret = ctf_merge_add_sou(cmp, i, B_TRUE);
682             if (ret != 0)
683                 return (ret);
684         }
685     }
686
687     /* Pass 2 */
688     for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
689         if (cmp->cm_tmap[i].cmt_missing == B_TRUE) {
690             ret = ctf_merge_add_type(cmp, i);
691             if (ret != 0)
692                 ctf_dprintf("Failed to merge type %d\n", i);
693             return (ret);
694         }
695     }
696
697     ret = ctf_update(cmp->cm_out);
698     if (ret != 0)
699         return (ret);
700
701     if (cmp->cm_dedup == B_TRUE) {
702         ctf_merge_fixup_dedup_map(cmp);
703     }
704
705     ctf_dprintf("Beginning merge pass 3\n");
706     /* Pass 3 */
707     for (i = 1; i <= cmp->cm_src->ctf_typemax; i++) {
708         if (cmp->cm_tmap[i].cmt_fixup == B_TRUE) {
709             ret = ctf_merge_fixup_type(cmp, i);
710             if (ret != 0)
711                 return (ret);
712         }
713     }
714
715     if (cmp->cm_dedup == B_TRUE) {
716         ctf_merge_fixup_dedup_map(cmp);
717     }
718
719     return (0);
720 }
721


---


722 unchanged_portion_omitted
723
724 /*
725  * After performing a pass, we need to go through the object and function type
726  * maps and potentially fix them up based on the new maps that we have.
727  */
728 static void
729 ctf_merge_fixup_symmaps(ctf_merge_types_t *cmp, ctf_merge_input_t *cmi)
730 {
731     ctf_merge_objmap_t *cmo;
732     ctf_merge_funcmap_t *cmf;
733
734     for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
735          cmo = list_next(&cmi->cmi_omap, cmo)) {
736         VERIFY3S(cmo->cmo_tid, !=, 0);
737         VERIFY(cm->cm_tmap[cmo->cmo_tid].cmt_map != 0);
738         cmo->cmo_tid = cm->cm_tmap[cmo->cmo_tid].cmt_map;
739     }
740
741     for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
742

```

```

795         cmf = list_next(&cmi->cmi_fmap, cmf)) {
796         int i;
797
798         VERIFY(cm->cm_tmap[cmf->cmf_rtid].cmt_map != 0);
799         cmf->cmf_rtid = cm->cm_tmap[cmf->cmf_rtid].cmt_map;
800         for (i = 0; i < cmf->cmf_args; i++) {
801             VERIFY(cm->cm_tmap[cmf->cmf_args[i]].cmt_map != 0);
802             cmf->cmf_args[i] =
803                 cm->cm_tmap[cmf->cmf_args[i]].cmt_map;
804         }
805     }
806
807     /*
808      * Merge the types contained inside of two input files. The second input file is
809      * always going to be the destination. We're guaranteed that it's always
810      * writeable.
811      */
812     static int
813     ctf_merge_types(void *arg, void *arg2, void **outp, void *unsued)
814 {
815     int ret;
816     ctf_merge_types_t cm;
817     ctf_diff_t *cdp;
818     ctf_merge_objmap_t *cmo;
819     ctf_merge_funcmap_t *cmf;
820     ctf_merge_input_t *scmi = arg;
821     ctf_merge_input_t *dcmi = arg2;
822     ctf_file_t *out = dcmi->cmi_input;
823     ctf_file_t *source = scmi->cmi_input;
824
825     ctf_dprintf("merging %p->%p\n", source, out);
826
827     if (!(out->ctf_flags & LCTF_RDWR))
828         return (ctf_set_errno(out, ECTF_RDONLY));
829
830     if (ctf_getmodel(out) != ctf_getmodel(source))
831         return (ctf_set_errno(out, ECTF_DMODEL));
832
833     if ((ret = ctf_diff_init(out, source, &cdp)) != 0)
834         return (ret);
835
836     cm.cm_out = out;
837     cm.cm_src = source;
838     cm.cm_dedup = B_FALSE;
839     cm.cm_unique = B_FALSE;
840     ret = ctf_merge_types_init(&cm);
841     if (ret != 0) {
842         ctf_diff_fini(cdp);
843         return (ctf_set_errno(out, ret));
844     }
845
846     ret = ctf_diff_types(cdp, ctf_merge_diffcb, &cm);
847     if (ret != 0)
848         goto cleanup;
849     ret = ctf_merge_common(&cm);
850     ctf_dprintf("merge common returned with %d\n", ret);
851     if (ret == 0) {
852         ret = ctf_update(out);
853         ctf_dprintf("update returned with %d\n", ret);
854     } else {
855         goto cleanup;
856     }
857
858     /*
859      * Now we need to fix up the object and function maps.
860      */

```

new/usr/src/lib/libctf/common/ctf\_merge.c

5

```

859 */
860 ctf_merge_fixup_symmaps(&cm, scmi);
820 for (cmo = list_head(&scmi->cmi_omap); cmo != NULL;
821      cmo = list_next(&scmi->cmi_omap, cmo)) {
822     if (cmo->cmo_tid == 0)
823       continue;
824     VERIFY(cm.cm_tmap[cmo->cmo_tid].cmt_map != 0);
825     cmo->cmo_tid = cm.cm_tmap[cmo->cmo_tid].cmt_map;
826   }
827
828 for (cmf = list_head(&scmi->cmi_fmap); cmf != NULL;
829      cmf = list_next(&scmi->cmi_fmap, cmf)) {
830   int i;
831
832   VERIFY(cm.cm_tmap[cmf->cmf_rtid].cmt_map != 0);
833   cmf->cmf_rtid = cm.cm_tmap[cmf->cmf_rtid].cmt_map;
834   for (i = 0; i < cmf->cmf_argc; i++) {
835     VERIFY(cm.cm_tmap[cmf->cmf_args[i]].cmt_map != 0);
836     cmf->cmf_args[i] = cm.cm_tmap[cmf->cmf_args[i]].cmt_map;
837   }
838 }
839
840 /*
841  * Now that we've fixed things up, we need to give our function and
842  * object maps to the destination, such that it can continue to update
843  * them going forward.
844  */
845 list_move_tail(&dcmi->cmi_fmap, &scmi->cmi_fmap);
846 list_move_tail(&dcmi->cmi_omap, &scmi->cmi_omap);
847
848 cleanup:
849   if (ret == 0)
850     *outp = dcmi;
851   ctf_merge_types_fini(&cm);
852   ctf_diff_fini(cdp);
853   if (ret != 0)
854     return (ctf_errno(out));
855   ctf_phase_bump();
856   return (0);
857 }
858 */
859 /* After performing a pass, we need to go through the object and function type
860  * maps and potentially fix them up based on the new maps that we have.
861 */
862 static void
863 ctf_merge_fixup_nontypes(ctf_merge_types_t *cmp, ctf_merge_input_t *cmi)
864 {
865   ctf_merge_objmap_t *cmo;
866   ctf_merge_funcmap_t *cmf;
867
868   for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
869        cmo = list_next(&cmi->cmi_omap, cmo)) {
870     if (cmo->cmo_tid == 0)
871       continue;
872     VERIFY(cmp->cm_tmap[cmo->cmo_tid].cmt_map != 0);
873     cmo->cmo_tid = cmp->cm_tmap[cmo->cmo_tid].cmt_map;
874   }
875
876   for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
877      cmf = list_next(&cmi->cmi_fmap, cmf)) {
878     int i;
879
880     VERIFY(cmp->cm_tmap[cmf->cmf_rtid].cmt_map != 0);
881     cmf->cmf_rtid = cmp->cm_tmap[cmf->cmf_rtid].cmt_map;
882     for (i = 0; i < cmf->cmf_argc; i++) {

```

new/usr/src/lib/libctf/common/ctf\_merge.c

```

883     VERIFY(cmp->cm_tmap[cmf->cmf_args[i]].cmt_map != 0);
884     cmf->cmf_args[i] = cmp->cm_tmap[cmf->cmf_args[i]].cmt_map;
885 }
886 }
887 }
888 }

889 }

890 static int
891 ctf_uniquify_types(ctf_merge_t *cmh, ctf_file_t *src, ctf_file_t **outp)
892 {
893     int err, ret;
894     ctf_file_t *out;
895     ctf_merge_types_t cm;
896     ctf_diff_t *cdp;
897     ctf_merge_input_t *cmi;
898     ctf_file_t *parent = cmh->cmh_unique;
899

900     *outp = NULL;
901     out = ctf_fdcREATE(cmh->cmh_ofd, &err);
902     if (out == NULL)
903         return (ctf_set_errno(src, err));
904

905     out->ctf_pname = cmh->cmh_pname;
906     if (ctf_setmodel(out, ctf_getmodel(parent)) != 0) {
907         (void) ctf_set_errno(src, ctf_errno(out));
908         ctf_close(out);
909         return (CTF_ERR);
910     }
911

912     if (ctf_import(out, parent) != 0) {
913         (void) ctf_set_errno(src, ctf_errno(out));
914         ctf_close(out);
915         return (ctf_set_errno(src, ctf_errno(parent)));
916     }
917

918     cm.cm_out = parent;
919     cm.cm_src = src;
920     cm.cm_dedup = B_FALSE;
921     cm.cm_unique = B_TRUE;
922     ret = ctf_merge_types_init(&cm);
923     if (ret != 0) {
924         ctf_close(out);
925         ctf_diff_fini(cdp);
926         return (ctf_set_errno(src, ret));
927     }
928

929     ret = ctf_diff_types(cdp, ctf_merge_diffcb, &cm);
930     if (ret == 0) {
931         cm.cm_out = out;
932         ret = ctf_merge_uniquify_types(&cm);
933         if (ret == 0)
934             ret = ctf_update(out);
935     }
936

937     if (ret != 0) {
938         ctf_merge_types_fini(&cm);
939         ctf_diff_fini(cdp);
940         return (ctf_set_errno(src, ctf_errno(cm.cm_out)));
941     }

```

```

939         for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
940             cmi = list_next(&cmh->cmh_inputs, cmi)) {
941             ctf_merge_fixup_symmaps(&cm, cmi);
942             ctf_merge_fixup_nontypes(&cm, cmi);
943         }
944
945         ctf_merge_types_fini(&cm);
946         ctf_diff_fini(cdp);
947         *outp = out;
948     }
949
950     return (0);
951
952     unchanged portion omitted
953
954 static int
955 ctf_merge_add_function(ctf_merge_input_t *cmi, ctf_funcinfo_t *fi, ulong_t idx,
956                         const char *file, const char *name, const Elf64_Sym *symp)
957 {
958     ctf_merge_input_t *cmi = arg;
959     ctf_merge_funcmap_t *fmap;
960
961     fmap = ctf_alloc(sizeof (ctf_merge_funcmap_t) +
962                      sizeof (ctf_id_t) * fi->ctc_argc);
963     if (fmap == NULL)
964         return (ENOMEM);
965
966     fmap->cmf_idx = idx;
967     fmap->cmf_sym = *symp;
968     fmap->cmf_rtid = fi->ctc_return;
969     fmap->cmf_flags = fi->ctc_flags;
970     fmap->cmf_argc = fi->ctc_argc;
971     fmap->cmf_name = name;
972
973     if (ELF64_ST_BIND(symp->st_info) == STB_LOCAL) {
974         fmap->cmf_file = file;
975     } else {
976         fmap->cmf_file = NULL;
977     }
978
979     if (ctf_func_args(cmi->cmi_input, idx, fmap->cmf_argc,
980                       fmap->cmf_args) != 0) {
981         ctf_free(fmap, sizeof (ctf_merge_funcmap_t) +
982                  sizeof (ctf_id_t) * fi->ctc_argc);
983         return (ctf_errno(cmi->cmi_input));
984     }
985
986     ctf_dprintf("added initial function %s, %lu, %s %u\n", name, idx,
987                 fmap->cmf_file != NULL ? fmap->cmf_file : "global",
988                 ELF64_ST_BIND(symp->st_info));
989     list_insert_tail(&cmi->cmi_fmap, fmap);
990
991     return (0);
992
993 static int
994 ctf_merge_add_object(ctf_merge_input_t *cmi, ctf_id_t id, ulong_t idx,
995                      const char *file, const char *name, const Elf64_Sym *symp)
996 {
997     ctf_merge_input_t *cmi = arg;
998     ctf_merge_objmap_t *cmo;
999
1000    cmo = ctf_alloc(sizeof (ctf_merge_objmap_t));
1001    if (cmo == NULL)
1002        return (ENOMEM);
1003
1004    cmo->cmo_name = name;

```

```

1005
1006         if (ELF64_ST_BIND(symp->st_info) == STB_LOCAL) {
1007             cmo->cmo_file = file;
1008         } else {
1009             cmo->cmo_file = NULL;
1010         }
1011         cmo->cmo_idx = idx;
1012         cmo->cmo_rtid = id;
1013         cmo->cmo_sym = *symp;
1014         list_insert_tail(&cmi->cmi omap, cmo);
1015
1016         ctf_dprintf("added initial object %s, %lu, %ld, %s\n", name, idx, id,
1017                     cmo->cmo_file != NULL ? cmo->cmo_file : "global");
1018
1019     return (0);
1020
1021     unchanged portion omitted
1022
1023 static int
1024 ctf_merge_add_symbol(const Elf64_Sym *symp, ulong_t idx, const char *file,
1025                      const char *name, boolean_t primary, void *arg)
1026 {
1027     ctf_merge_input_t *cmi = arg;
1028     ctf_file_t *fp = cmi->cmi_input;
1029     ushort_t *data, funcbase;
1030     uint_t type;
1031     ctf_funcinfo_t fi;
1032
1033     /*
1034      * See if there is type information for this. If there is no
1035      * type information for this entry or no translation, then we
1036      * will find the value zero. This indicates no type ID for
1037      * objects and encodes unknown information for functions.
1038      */
1039     if (fp->ctf_sxlate[idx] == -1u)
1040         return (0);
1041     data = (ushort_t *)((uintptr_t)fp->ctf_buf + fp->ctf_sxlate[idx]);
1042     if (*data == 0)
1043         return (0);
1044
1045     type = ELF64_ST_TYPE(symp->st_info);
1046
1047     switch (type) {
1048     case STT_FUNC:
1049         funcbase = *data;
1050         if (LCTF_INFO_KIND(fp, funcbase) != CTF_K_FUNCTION)
1051             return (0);
1052         data++;
1053         fi.ctc_return = *data;
1054         data++;
1055         fi.ctc_argc = LCTF_INFO_VLEN(fp, funcbase);
1056         fi.ctc_flags = 0;
1057
1058         if (fi.ctc_argc != 0 && data[fi.ctc_argc - 1] == 0) {
1059             fi.ctc_flags |= CTF_FUNC_VARARG;
1060             fi.ctc_argc--;
1061         }
1062         return (ctf_merge_add_function(cmi, &fi, idx, file, name,
1063                                       symp));
1064     case STT_OBJECT:
1065         return (ctf_merge_add_object(cmi, *data, idx, file, name,
1066                                     symp));
1067     default:
1068         return (0);
1069     }
1070
1071     unchanged portion omitted
1072
1073     /*
```

new/usr/src/lib/libctf/common/ctf\_merge.c

```

1166 * Whenever we create an entry to merge, we then go and add a second empty
1167 * ctf_file_t which we use for the purposes of our merging. It's not the best,
1168 * but it's the best that we've got at the moment.
1169 */
1170 int
1171 ctf_merge_add(ctf_merge_t *cmh, ctf_file_t *input)
1172 {
1173     int ret;
1174     ctf_merge_input_t *cmi;
1175     ctf_file_t *empty;
1176
1177     ctf_dprintf("adding input %p\n", input);
1178
1179     if (input->ctf_flags & LCTF_CHILD)
1180         return (ECTF_MCHILD);
1181
1182     cmi = ctf_alloc(sizeof (ctf_merge_input_t));
1183     if (cmi == NULL)
1184         return (ENOMEM);
1185
1186     cmi->cmi_created = B_FALSE;
1187     cmi->cmi_input = input;
1188     list_create(&cmi->cmi_fmap, sizeof (ctf_merge_funcmap_t),
1189                 offsetof(ctf_merge_funcmap_t, cmf_node));
1190     list_create(&cmi->cmi_omap, sizeof (ctf_merge_funcmap_t),
1191                 offsetof(ctf_merge_objmap_t, cmo_node));
1192
1193     if (cmh->cmh_msyms == B_TRUE) {
1194         if ((ret = ctf_symtab_iter(input, ctf_merge_add_symbol,
1195                                     if ((ret = ctf_function_iter(input, ctf_merge_add_funcs_cb,
1196                                         cmi)) != 0) {
1197                                         ctf_merge_fini_input(cmi);
1198                                         return (ret);
1199         }
1200
1201         if ((ret = ctf_object_iter(input, ctf_merge_add_objs_cb,
1202                                     cmi)) != 0) {
1203                                         ctf_merge_fini_input(cmi);
1204                                         return (ret);
1205         }
1206
1207         list_insert_tail(&cmh->cmh_inputs, cmi);
1208         cmh->cmh_ninputs++;
1209
1210         /* And now the empty one to merge into this */
1211         cmi = ctf_alloc(sizeof (ctf_merge_input_t));
1212         if (cmi == NULL)
1213             return (ENOMEM);
1214         list_create(&cmi->cmi_fmap, sizeof (ctf_merge_funcmap_t),
1215                     offsetof(ctf_merge_funcmap_t, cmf_node));
1216         list_create(&cmi->cmi_omap, sizeof (ctf_merge_funcmap_t),
1217                     offsetof(ctf_merge_objmap_t, cmo_node));
1218
1218         empty = ctf_fdcreate(cmh->cmh_ofd, &ret);
1219         if (empty == NULL)
1220             return (ret);
1221         cmi->cmi_input = empty;
1222         cmi->cmi_created = B_TRUE;
1223
1224         if (ctf_setmodel(empty, ctf_getmodel(input)) == CTF_ERR) {
1225             return (ctf_errno(empty));
1226         }
1227
1228         list_insert_tail(&cmh->cmh_inputs, cmi);
1229         cmh->cmh_ninputs++;

```

new/usr/src/lib/libctf/common/ctf\_merge.c

```

1225         ctf_dprintf("added containers %p and %p\n", input, empty);
1226         return (0);
1227     }
1228     unchanged_portion_omitted
1229
1230     /*
1231      * Symbol matching rules: the purpose of this is to verify that the type
1232      * information that we have for a given symbol actually matches the output
1233      * symbol. This is unfortunately complicated by several different factors:
1234      *
1235      * 1. When merging multiple .o's into a single item, the symbol table index will
1236      * not match.
1237      *
1238      * 2. Visibility of a symbol may not be identical to the object file or the
1239      * DWARF information due to symbol reduction via a mapfile.
1240      *
1241      * As such, we have to employ the following rules:
1242      *
1243      * 1. A global symbol table entry always matches a global CTF symbol with the
1244      * same name.
1245      *
1246      * 2. A local symbol table entry always matches a local CTF symbol if they have
1247      * the same name and they belong to the same file.
1248      *
1249      * 3. A weak symbol matches a non-weak symbol. This happens if we find that the
1250      * types match, the values match, the sizes match, and the section indexes
1251      * match. This happens when we do a conversion in one pass, it almost never
1252      * happens when we're merging multiple object files. If we match a CTF global
1253      * symbol, that's a fixed match, otherwise it's a fuzzy match.
1254      *
1255      * 4. A local symbol table entry matches a global CTF entry if the
1256      * other pieces fail, but they have the same name. This is considered a fuzzy
1257      * match and is not used unless we have no other options.
1258      *
1259      * 5. A weak symbol table entry matches a weak CTF entry if the other pieces
1260      * fail, but they have the same name. This is considered a fuzzy match and is
1261      * not used unless we have no other options. When merging independent .o files,
1262      * this is often the only recourse we have to matching weak symbols.
1263      *
1264      * In the end, this would all be much simpler if we were able to do this as part
1265      * of libld which would be able to do all the symbol transformations.
1266      */
1267
1268     static boolean_t
1269     ctf_merge_symbol_match(const char *ctf_file, const char *ctf_name,
1270                           const Elf64_Sym *ctf_symp, const char *syntab_file, const char *syntab_name,
1271                           const Elf64_Sym *syntab_symp, boolean_t *is_fuzzy)
1272
1273     static int
1274     ctf_merge_symbols(ctf_merge_t *cmh, ctf_file_t *fp)
1275     {
1276         *is_fuzzy = B_FALSE;
1277         uint_t syntab_bind, ctf_bind;
1278         int err;
1279         ulong_t i;
1280
1281         syntab_bind = ELF64_ST_BIND(syntab_symp->st_info);
1282         ctf_bind = ELF64_ST_BIND(ctf_symp->st_info);
1283         uintptr_t symbase = (uintptr_t)fp->ctf_syntab.cts_data;
1284         uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;
1285
1286         ctf_dprintf("comparing merge match for %s/%s/%u->%s/%s/%u\n",
1287                     syntab_file, syntab_name, syntab_bind,
1288                     ctf_file, ctf_name, ctf_bind);
1289         if (strcmp(ctf_name, syntab_name) != 0) {
1290             return (B_FALSE);
1291         }
1292         for (i = 0; i < fp->ctf_nsyms; i++) {
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3489
3490
3491
3492
3493
349
```

new/usr/src/lib/libctf/common/ctf\_merge.c

11

```

1207     const char *name;
1208     ctf_merge_input_t *cmi;
1209     ctf_merge_objmap_t *cmo;
1210
1211     if (symtab_bind == STB_GLOBAL && ctf_bind == STB_GLOBAL) {
1212         return (B_TRUE);
1213     } else if (symtab_bind == STB_GLOBAL) {
1214         return (B_FALSE);
1215         if (fp->ctf_symtab.cts_entsize == sizeof (Elf32_Sym)) {
1216             const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
1217             int type = ELF32_ST_TYPE(symp->st_info);
1218             if (type != STT_OBJECT)
1219                 continue;
1220             if (ctf_sym_valid(strbase, type, symp->st_shndx,
1221                             symp->st_value, symp->st_name) == B_FALSE)
1222                 continue;
1223             name = (char *)(strbase + symp->st_name);
1224         } else {
1225             const Elf64_Sym *symp = (Elf64_Sym *)symbase + i;
1226             int type = ELF64_ST_TYPE(symp->st_info);
1227             if (type != STT_OBJECT)
1228                 continue;
1229             if (ctf_sym_valid(strbase, type, symp->st_shndx,
1230                             symp->st_value, symp->st_name) == B_FALSE)
1231                 continue;
1232             name = (char *)(strbase + symp->st_name);
1233         }
1234
1235     if (ctf_bind == STB_LOCAL && ctf_bind == symtab_bind &&
1236         ctf_file != NULL && symtab_file != NULL &&
1237         strcmp(ctf_file, symtab_file) == 0) {
1238         return (B_TRUE);
1239         cmo = NULL;
1240         for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
1241             cmi = list_next(&cmh->cmh_inputs, cmi)) {
1242             for (cmo = list_head(&cmi->cmi_omap); cmo != NULL;
1243                 cmo = list_next(&cmi->cmi_omap, cmo)) {
1244                 if (strcmp(cmo->cmo_name, name) == 0)
1245                     goto found;
1246         }
1247
1248     if (symtab_bind == STB_WEAK && ctf_bind != STB_WEAK &&
1249         ELF64_ST_TYPE(symtab_symp->st_info) ==
1250         ELF64_ST_TYPE(ctf_symp->st_info) &&
1251         symtab_symp->st_value == ctf_symp->st_value &&
1252         symtab_symp->st_size == ctf_symp->st_size &&
1253         symtab_symp->st_shndx == ctf_symp->st_shndx) {
1254         if (ctf_bind == STB_GLOBAL) {
1255             return (B_TRUE);
1256         }
1257
1258     if (ctf_bind == STB_LOCAL && ctf_file != NULL &&
1259         symtab_file != NULL && strcmp(ctf_file, symtab_file) == 0) {
1260         *is_fuzzy = B_TRUE;
1261         return (B_TRUE);
1262     }
1263     if (cmo != NULL) {
1264         if (cmo->cmo_tid == 0)
1265             continue;
1266         if ((err = ctf_add_object(fp, i, cmo->cmo_tid)) != 0)
1267             ctf_dprintf("Failed to add symbol %s->%d: %s\n",
1268                         name, cmo->cmo_tid,
1269                         ctf_errmsg(ctf_errno(fp)));
1270         return (err);
1271     }
1272 }

```

new/usr/src/lib/libctf/common/ctf\_merge.c

```
1334     if (ctf_bind == STB_GLOBAL ||
1335         (ctf_bind == STB_WEAK && symtab_bind == STB_WEAK)) {
1336         *is_fuzzy = B_TRUE;
1337         return (B_TRUE);
1338     }
1340     return (B_FALSE);
1253     return (0);
1341 }
1343 /*
1344  * For each symbol, try and find a match. We will attempt to find an exact
1345  * match; however, we will settle for a fuzzy match in general. There is one
1346  * case where we will not opt to use a fuzzy match, which is when performing the
1347  * deduplication of a container. In such a case we are trying to reduce common
1348  * types and a fuzzy match would be inappropriate as if we're in the context of
1349  * a single container, the conversion process should have identified any exact
1350  * or fuzzy matches that were required.
1351 */
1352 static int
1353 ctf_merge_symbols(const Elf64_Sym *symp, ulong_t idx, const char *file,
1354 const char *name, boolean_t primary, void *arg)
1257 ctf_merge_functions(ctf_merge_t *cmh, ctf_file_t *fp)
1355 {
1356     int err;
1357     uint_t type, bind;
1358     ctf_merge_symbol_arg_t *csa = arg;
1359     ctf_file_t *fp = csa->cmsa_out;
1260     ulong_t i;
1261     ctf_funcinfo_t fi;
1361     type = ELF64_ST_TYPE(symp->st_info);
1362     bind = ELF64_ST_BIND(symp->st_info);
1263     uintptr_t symbase = (uintptr_t)fp->ctf_symtab.cts_data;
1264     uintptr_t strbase = (uintptr_t)fp->ctf_strtab.cts_data;
1364     ctf_dprintf("Trying to find match for %s/%s/%u\n", file, name,
1365                 ELF64_ST_BIND(symp->st_info));
1266     for (i = 0; i < fp->ctf_nsyms; i++) {
1267         const char *name;
1268         ctf_merge_input_t *cmi;
1269         ctf_merge_funcmap_t *cmf;
1367         if (type == STT_OBJECT) {
1368             ctf_merge_objmap_t *cmo, *match = NULL;
1370             for (cmo = list_head(csa->cmsa_objmap); cmo != NULL;
1371                  cmo = list_next(csa->cmsa_objmap, cmo)) {
1372                 boolean_t is_fuzzy = B_FALSE;
1373                 if (ctf_merge_symbol_match(cmo->cmo_file, cmo->cmo_name,
1374                     &cmo->cmo_sym, file, name, symp, &is_fuzzy)) {
1375                     if (is_fuzzy && csa->cmsa_dedup &&
1376                         bind != STB_WEAK) {
1271                     if (fp->ctf_symtab.cts_entsize == sizeof(Elf32_Sym)) {
1272                         const Elf32_Sym *symp = (Elf32_Sym *)symbase + i;
1273                         int type = ELF32_ST_TYPE(symp->st_info);
1274                         if (ELF32_ST_TYPE(symp->st_info) != STT_FUNC)
1377                             continue;
1378                     }
1379                     match = cmo;
1380                     if (is_fuzzy) {
1276                     if (ctf_sym_valid(strbase, type, symp->st_shndx,
1277                         symp->st_value, symp->st_name) == B_FALSE)
1381                         continue;
1382                 }
1383             }
1384         }
1385     }
1386 }
```

```

1383         break;
1384     }
1385 }
1386 if (match == NULL) {
1387     return (0);
1388 }
1389 if ((err = ctf_add_object(fp, idx, match->cmo_tid)) != 0) {
1390     ctf_dprintf("Failed to add symbol %s->%d: %s\n", name,
1391                 match->cmo_tid, ctf_errmsg(ctf_errno(fp)));
1392     return (ctf_errno(fp));
1393 }
1394 ctf_dprintf("mapped object into output %s/%s->%ld\n", file,
1395             name, match->cmo_tid);
1396             name = (char *)strbase + symp->st_name);
1397 } else {
1398     ctf_merge_funcmap_t *cmf, *match = NULL;
1399     ctf_funcinfo_t fi;
1400
1401     for (cmf = list_head(csa->cmsa_funcmap); cmf != NULL;
1402          cmf = list_next(csa->cmsa_funcmap, cmf)) {
1403         boolean_t is_fuzzy = B_FALSE;
1404         if (ctf_merge_symbol_match(cmf->cmf_file, cmf->cmf_name,
1405             &cmf->cmf_sym, file, name, symp, &is_fuzzy)) {
1406             if (is_fuzzy && csa->cmsa_dedup &
1407                 bind != STB_WEAK) {
1408                 const Elf64_Sym *symp = (Elf64_Sym *)sympbase + i;
1409                 int type = ELF64_ST_TYPE(symp->st_info);
1410                 if (ELF64_ST_TYPE(symp->st_info) != STT_FUNC)
1411                     continue;
1412                 match = cmf;
1413                 if (is_fuzzy) {
1414                     if (ctf_sym_valid(strbase, type, symp->st_shndx,
1415                         symp->st_value, symp->st_name) == B_FALSE)
1416                         continue;
1417                     name = (char *)strbase + symp->st_name);
1418                 }
1419             }
1420         }
1421         if (match == NULL) {
1422             return (0);
1423             cmf = NULL;
1424         }
1425         for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
1426              cmi = list_next(&cmh->cmh_inputs, cmi)) {
1427             for (cmf = list_head(&cmi->cmi_fmap); cmf != NULL;
1428                 cmf = list_next(&cmi->cmi_fmap, cmf)) {
1429                 if (strcmp(cmf->cmf_name, name) == 0)
1430                     goto found;
1431             }
1432             fi.ctc_return = match->cmf_rtid;
1433             fi.ctc_argc = match->cmf_argc;
1434             fi.ctc_flags = match->cmf_flags;
1435             if ((err = ctf_add_function(fp, idx, &fi, match->cmf_args)) != 0) {
1436                 ctf_dprintf("Failed to add function %s: %s\n", name,
1437                             ctf_errmsg(ctf_errno(fp)));
1438                 return (ctf_errno(fp));
1439             }
1440             ctf_dprintf("mapped function into output %s/%s\n", file,
1441                         name);
1442     found:
1443

```

```

1301     if (cmf != NULL) {
1302         fi.ctc_return = cmf->cmf_rtid;
1303         fi.ctc_argc = cmf->cmf_argc;
1304         fi.ctc_flags = cmf->cmf_flags;
1305         if ((err = ctf_add_function(fp, i, &fi,
1306             cmf->cmf_args)) != 0)
1307             return (err);
1308     }
1309 }
1310
1311 return (0);
1312 }
1313
1314 int
1315 ctf_merge_merge(ctf_merge_t *cmh, ctf_file_t **outp)
1316 {
1317     int err, merr;
1318     ctf_merge_input_t *cmi;
1319     ctf_id_t ltype;
1320     mergeq_t *mfp;
1321     ctf_merge_input_t *final;
1322     ctf_file_t *out;
1323
1324 ctf_dprintf("Beginning ctf_merge_merge()\n");
1325 if (cmh->cmh_label != NULL && cmh->cmh_unique != NULL) {
1326     const char *label = ctf_label_topmost(cmh->cmh_unique);
1327     if (label == NULL)
1328         return (ECTF_NOLABEL);
1329     if (strcmp(label, cmh->cmh_label) != 0)
1330         return (ECTF_LCONFLICT);
1331 }
1332
1333 if (mergeq_init(&mfp, cmh->cmh_nthreads) == -1) {
1334     return (errno);
1335 }
1336
1337 VERIFY(cmh->cmh_ninputs % 2 == 0);
1338 for (cmi = list_head(&cmh->cmh_inputs); cmi != NULL;
1339      cmi = list_next(&cmh->cmh_inputs, cmi)) {
1340     if (mergeq_add(mfp, cmi) == -1) {
1341         err = errno;
1342         mergeq_fini(mfp);
1343     }
1344 }
1345
1346 err = mergeq_merge(mfp, ctf_merge_types, NULL, (void **)&final, &merr);
1347 mergeq_fini(mfp);
1348
1349 if (err == MERGEQ_ERROR) {
1350     return (errno);
1351 } else if (err == MERGEQ_UERROR) {
1352     return (merr);
1353 }
1354
1355 /*
1356  * Disassociate the generated ctf_file_t from the original input. That
1357  * way when the input gets cleaned up, we don't accidentally kill the
1358  * final reference to the ctf_file_t. If it gets uniquified then we'll
1359  * kill it.
1360 */
1361 VERIFY(final->cmi_input != NULL);
1362 out = final->cmi_input;
1363 final->cmi_input = NULL;
1364
1365 ctf_dprintf("preparing to uniquify against: %p\n", cmh->cmh_unique);

```

```

1491     if (cmh->cmh_unique != NULL) {
1492         ctf_file_t *u;
1493         err = ctf_uniquify_types(cmh, out, &u);
1494         if (err != 0) {
1495             err = ctf_errno(out);
1496             ctf_close(out);
1497             return (err);
1498         }
1499         ctf_close(out);
1500         out = u;
1501     }
1503     ltype = out->ctf_typemax;
1504     if ((out->ctf_flags & LCTF_CHILD) && ltype != 0)
1505         ltype += CTF_CHILD_START;
1506     ctf_dprintf("trying to add the label\n");
1507     if (cmh->cmh_label != NULL &&
1508         ctf_add_label(out, cmh->cmh_label, ltype, 0) != 0) {
1509         ctf_close(out);
1510         return (ctf_errno(out));
1511     }
1513     ctf_dprintf("merging symbols and the like\n");
1514     if (cmh->cmh_msyms == B_TRUE) {
1515         ctf_merge_symbol_arg_t arg;
1516         arg.cmsa_objmap = &final->cmi_omap;
1517         arg.cmsa_funcmap = &final->cmi_fmap;
1518         arg.cmsa_out = out;
1519         arg.cmsa_dedup = B_FALSE;
1520         err = ctf_symtab_iter(out, ctf_merge_symbols, &arg);
1521         err = ctf_merge_symbols(cmh, out);
1522         if (err != 0) {
1523             ctf_close(out);
1524             return (err);
1525             return (ctf_errno(out));
1526         }
1527         err = ctf_merge_functions(cmh, out);
1528         if (err != 0) {
1529             ctf_close(out);
1530             return (err);
1531             return (ctf_errno(out));
1532         }
1534         *outp = out;
1535         return (0);
1536     }
1569 */
1570 * Dedup a CTF container.
1571 *
1572 * DWARF and other encoding formats that we use to create CTF data may create
1573 * multiple copies of a given type. However, after doing a conversion, and
1574 * before doing a merge, we'd prefer, if possible, to have every input container
1575 * to be unique.
1576 *
1577 * Doing a deduplication is like a normal merge. However, when we diff the types

```

```

1578     * in the container, rather than doing a normal diff, we instead want to diff
1579     * against any already processed types. eg, for a given type i in a container,
1580     * we want to diff it from 0 to i - 1.
1581     */
1582     int
1583     ctf_merge_dedup(ctf_merge_t *cmp, ctf_file_t **outp)
1584     {
1585         int ret;
1586         ctf_diff_t *cdp = NULL;
1587         ctf_merge_input_t *cmi, *cmc;
1588         ctf_file_t *ifp, *ofp;
1589         ctf_merge_types_t cm;
1591         if (cmp == NULL || outp == NULL)
1592             return (EINVAL);
1594         ctf_dprintf("encountered %d inputs\n", cmp->cmh_ninputs);
1595         if (cmp->cmh_ninputs != 2)
1596             return (EINVAL);
1598         ctf_dprintf("passed argument sanity check\n");
1600         cmi = list_head(&cmp->cmh_inputs);
1601         VERIFY(cmi != NULL);
1602         cmc = list_next(&cmp->cmh_inputs, cmi);
1603         VERIFY(cmc != NULL);
1604         ifp = cmi->cmi_input;
1605         ofp = cmc->cmi_input;
1606         VERIFY(ifp != NULL);
1607         VERIFY(ofp != NULL);
1608         cm.cm_src = ifp;
1609         cm.cm_out = ofp;
1610         cm.cm_dedup = B_TRUE;
1611         cm.cm_unique = B_FALSE;
1613         if ((ret = ctf_merge_types_init(&cm)) != 0) {
1614             return (ret);
1615         }
1617         if ((ret = ctf_diff_init(ifp, ifp, &cdp)) != 0)
1618             goto err;
1620         ctf_dprintf("Successfully initialized dedup\n");
1621         if ((ret = ctf_diff_self(cdp, ctf_dedup_cb, &cm)) != 0)
1622             goto err;
1624         ctf_dprintf("Successfully diffed types\n");
1625         ret = ctf_merge_common(&cm);
1626         ctf_dprintf("deduping types result: %d\n", ret);
1627         if (ret == 0)
1628             ret = ctf_update(cm.cm_out);
1629         if (ret != 0)
1630             goto err;
1632         ctf_dprintf("Successfully deduped types\n");
1633         ctf_phase_dump(cm.cm_out, "dedup-pre-syms", NULL);
1634         ctf_phase_dump(cm.cm_out, "dedup-pre-syms");
1635         /*
1636         * Now we need to fix up the object and function maps.
1637         */
1638         ctf_merge_fixup_symmaps(&cm, cmi);
1639         ctf_merge_fixup_nontypes(&cm, cmi);
1640         if (cmp->cmh_msyms == B_TRUE) {
1641             ctf_merge_symbol_arg_t arg;

```

```
1642     arg.cmsa_objmap = &cmi->cmi_omap;
1643     arg.cmsa_funcmap = &cmi->cmi_fmap;
1644     arg.cmsa_out = cm.cm_out;
1645     arg.cmsa_dedup = B_TRUE;
1646     ret = ctf_symtab_iter(cm.cm_out, ctf_merge_symbols, &arg);
1647     ret = ctf_merge_symbols(cmp, cm.cm_out);
1648     if (ret != 0) {
1649         ret = ctf_errno(cm.cm_out);
1650         ctf_dprintf("failed to dedup symbols: %s\n",
1651                     ctf_errmsg(ret));
1652         goto err;
1653     }
1654     ret = ctf_merge_functions(cmp, cm.cm_out);
1655     if (ret != 0) {
1656         ret = ctf_errno(cm.cm_out);
1657         ctf_dprintf("failed to dedup functions: %s\n",
1658                     ctf_errmsg(ret));
1659         goto err;
1660     }
1661     ret = ctf_update(cm.cm_out);
1662     if (ret == 0) {
1663         cmc->cmi_input = NULL;
1664     }
1665     *outp = cm.cm_out;
1666     ctf_phase_dump(cm.cm_out, "dedup-post-syms", NULL);
1667     err:
1668     ctf_merge_types_fini(&cm);
1669     ctf_diff_fini(cdp);
1670     return (ret);
1671 }
```

unchanged portion omitted

```
new/usr/src/lib/libctf/common/libctf_impl.h
```

```
*****
```

```
1648 Tue Apr 23 05:24:02 2019
```

```
new/usr/src/lib/libctf/common/libctf_impl.h
```

```
10812 ctf tools shouldn't add blank labels
```

```
10813 ctf symbol mapping needs work
```

```
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
```

```
*****
```

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License (" CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */
```

```
12 /*
13  * Copyright 2019 Joyent, Inc.
14  * Copyright 2015 Joyent, Inc.
```

```
16 #ifndef _LIBCTF_IMPL_H
17 #define _LIBCTF_IMPL_H

19 /*
20  * Portions of libctf implementations that are only suitable for CTF's userland
21  * library, eg. converting and merging related routines.
22 */

24 #include <libelf.h>
25 #include <libctf.h>
26 #include <ctf_impl.h>

28 #ifdef __cplusplus
29 extern "C" {
30 #endif

32 typedef enum ctf_conv_status {
33     CTF_CONV_SUCCESS      = 0,
34     CTF_CONV_ERROR        = 1,
35     CTF_CONV_NOTSUP       = 2
36 } ctf_conv_status_t;

38 typedef ctf_conv_status_t (*ctf_convert_f)(int, Elf *, uint_t, int *,
39                                         ctf_file_t **, char *, size_t);
40 extern ctf_conv_status_t ctf_dwarf_convert(int, Elf *, uint_t, int *,
41                                         ctf_file_t **, char *, size_t);

43 /*
44  * Symbol walking
45 */
46 typedef int (*ctf_symtab_f)(const Elf64_Sym *, ulong_t, const char *,
47                           const char *, boolean_t, void *);
48 extern int ctf_symtab_iter(ctf_file_t *, ctf_symtab_f, void *);

50 /*
51  * zlib compression routines
52 */
53 extern int ctf_compress(ctf_file_t *fp, void **, size_t *, size_t *);

55 extern int ctf_diff_self(ctf_diff_t *, ctf_diff_type_f, void *);

57 /*
58  * Internal debugging aids
```

```
1
```

```
new/usr/src/lib/libctf/common/libctf_impl.h
```

```
59 */
60 extern void ctf_phase_dump(ctf_file_t *, const char *, const char *);
61 extern void ctf_phase_bump(void);
63 #ifdef __cplusplus
64 }
```

```
_____ unchanged_portion_omitted _____
```

```
2
```