Saját nyelvi felület készítése GCC-hez

Jó hír a nyelvtervezőknek hogy egyszerűbbé vált a nyelvi felületek készítése a GCC-hez.

A *GCC*, ez az elsőrangú ingyenes programfordító készlet, rengeteg változáson ment keresztül az utóbbi néhány évben. Az egyik ilyen változás, nevezetesen a ***tree-ssa*** ág beolvasztása lényegesen leegyszerűsítette az új *GCC* előtétek készítését.
A *GCC* mindig is két belső megvalósítással rendelkezett, a fákkal és az *RTL*-el. Az *RTL*, azaz regiszter átviteli nyelv, az alacsonyabb szintű megoldás, ezt használja a *GCC* a gépi kód készítésekor. Korábban minden optimalizálás az *RTL* szintjén folyt. A fák magasabb szintű megoldások. Hagyományosan ezek kevésbé dokumentáltak és kevésbé ismertek voltak mint az *RTL*.
A ***Diego Novillo*** vezette, *GCC* belső részeinek hosszú távú újraírását célzó ***tree-ssa*** projekt mindezt megváltoztatja.
A fák azóta sokat fejlődtek, igaz még mindig kicsit hiányosan dokumentáltak, viszont sok optimalizálás már a fa szinten megtörténik. A munka hasznos melléktermé a fa alapú nyelvek egyértelmű megfogalmazása: a *GENERIC*. Minden *GCC* nyelvi felület *GENERIC*-et készít, amelyet később egy másik, *GIMPLE* nevű alacsonyabb szintű fa alapú leírássá alakítunk, végül innen lépünk tovább a *RTL*-re. Ami ebből számunkra lényeges, hogy mostantól sokkal, de sokkal könnyebb nyelvi felületet írni a *GCC*-hez. Tulajdonképpen, most már akár az is képes *GCC* nyelvi felületet készíteni, aki egyáltalán nem ért az *RTL* megvalósításokhoz. Ez a cikk végigvezet bennünket azokon a lépéseken, melyekkel elkészíthetjük saját fordítóprogramunk *GCC* felületét. A cikkben leírtak a 2005-ben megjelent *GCC 4.0*-ás verzióra vonatkoznak.

A program ábrázolása

A mi szempontunkból a fordítás két részből áll, értelmezés és szemantikai vizsgálat, majd a kódgenerálás. A *GCC* a második részt elvégzi, így már csak az a kérdés hogyan tudjuk a legjobban megvalósítani az első fázist?
A hagyományos *GCC* nyelvi felületek, például a *C* és a *C++* felület, értelmezés közben készítik el a fát. Az ilyen felületek a nyelv specifikus szerkezetekhez saját fa-kódot adnak meg. Aztán amikor a szemantikai vizsgálat befejeződött, ezeket a fákat *GENERIC*-é alakítják, a magas szintű nyelvspecifikus fákat alacsonyabb szintű változatokkal helyettesítve. A megközelítés egyik előnye, hogy a nyelvspecifikus fák már amúgy is közel *GENERIC* alakúak. A szintcsökkentési fázissal gyakran elkerülhető, hogy túlságosan sok szemét keletkezzen.
A legfőbb probléma ezzel a megközelítéssel, hogy a fa dinamikus típusú. Elméletben ez még nem olyan nagy baj, hiszen rengeteg dinamikus típusú környezet létezik amelyeket hatékonyan használhatnak a fejlesztők, például ilyen a ***Lisp*** és a ***Python*** is. Ugyanakkor ezek teljes környezetek, a *GCC* erősen „makrósított" *C* kódjáról viszont nem lehet elmondani ugyanezeket az előnyöket.
Magam részéről azokat a megközelítéseket kedvelem, amelyek erősen típusos nyelvspecifikus programábrázolást más néven az úgynevezett ***„abstract syntax tree" (AST)*** megoldást alkalmazzák. Ezt a megközelítést alkalmazza az Ada felület és a Java programozási nyelvhez készült nyelvi felület újraírt változata, a *gcjx*.
A *gcjx C++* nyelven készült és a java programozási nyelv osztályait modellező osztály hierarchiával rendelkezik.
Ez a kód lényegében független a *GCC*-től és akár más célra is felhasználható. A *gcjx* esetében például a modellt levihetjük a *GENERIC* szintjére, de akár használhatjuk bájtkód vagy *JNI* fejlécfájlok előállítására is. Ezen kívül akár kódvizsgálatot is végezhetünk; a felület gyakorlatilag újrahasznosítható könyvtár lesz.
Ez a megközelítés az erősen típusos tervezés minden előnyét élvezi. A *GCC* esetében ez konkrétan annyit jelent, hogy könnyebben karbantartható és érthető programot kapunk. Az eredményül kapott felület és a *GCC* viszonylagos függetlensége azért is előnyös, mert a *GCC* igen gyorsan változik és ez a laza párosítás minimálisra csökkenti hiba lehetőségét.
A megközelítés nyilvánvaló hátránya hogy a fordítóprogramunknak valószínűleg több munkát kell végeznie mint amennyire szigorúan szüksége lenne, vagy több memóriát használ. A gyakorlatban ez nem tűnik igazán fontosnak.
Mielőtt belevágnánk a *GCC* felületkészítés részleteinek megismerésébe, nézzük meg milyen dokumentációkat és forrásfájlokat érdemes megismernünk. Tekintettel arra, hogy a *GCC* közösség nem tulajdonított túl nagy jelentőséget az egyszerűen elkészíthető nyelvi felületnek, néhány fontos dolog csak a forrásban van dokumentálva. Az itt felsorolt dokumentációk ***info*** lapokra és nem ***URL***-ekre hivatkoznak, hiszen a *GCC 4.0* még nem jött ki. Ezért aztán a weblapokon a korábbi verziókat láthatjuk. A legjobb amit tehetünk, hogy a *CVS*-ből letöltjük a *GCC*-t és átlapozzuk a forráskódot.

- *gcc/c.opt*: leírja a *C* családba tartozó nyelvi felületek parancssori kapcsolóit. És ami még fontosabb bemutatja az *.opt* állományok formátumát. Ilyet kell majd írnunk.

- *gcc* infó lap, ***Spec Files*** csomópont (***gcc/doc/invoke.texi*** forrásfájl): a *GCC* meghajtó által használt meghatározó mininyelvet írja le. Ilyen meghatározásokat fogunk készíteni, amely megmutatja a *GCC*-nek, hogyan kell meghívnia a felületünket.
- ***gccint*** infó lap, ***Front End*** csomópont (***gcc/doc/sourcebuild.texi*** forrásfájl): ismerteti hogyan integráljuk a felületünket a *GCC* fordítási folyamatába.
- ***gccint*** infó lap, ***Tree SSA*** csomópont (***gcc/doc/tree-ssa.texi*** forrásfájl): a *GENERIC* leírása.
- ***gcc/tree.def***, ***gcc/tree.h***: a fák néhány olyan jellemzője ami úgy tűnik kimaradt a dokumentációkból, ezeket a fájlokat elolvasva megismerhető. A ***tree.def*** definiálja a legtöbb fa kódot és tartalmaz néhány átfogó megjegyzést. A ***tree.h*** definiálja a fa szerkezetét, a sok elérési makrót valamint néhány függvényt, amelyek a különféle fák létrehozása során lehetnek segítségünkre.
- ***libcpp/include/line-map.h***: a sortérképet a *GCC* forráskód helyek tárolására használja. Ezt tetszés szerint használhatjuk a saját nyelvi felületünk megvalósításában. A ***gcjx*** nem használja. Ha nem is használjuk őket, a *GENERIC* szintjére lépéskor mindenképpen el kell készíteni azokat, ugyanis a sortérképben tárolt információkat használjuk a hibakeresési információk előállításához.
- ***gcc/errors.h***, ***gcc/diagnostic.h***: A *GCC* hibaformázási függvényeit definiálja ezeket tetszés szerint használhatjuk.
- ***gcc/gdbinit.in***: néhány *GDB* parancsot definiál amelyek jól jöhetnek ha hibát keresünk a *GCC*-ben. Például a `pt` parancs szöveges alakban rajzolja ki a fát. A *GCC* fordítási könyvtárában készül egy ***.gdbinit*** állomány, így ha hibát keresünk a makrók azonnal elérhetőek lesznek.
- ***gcc/langhooks.h***: a *GCC* a nyelvi horgok módszerének alkalmazásával teszi lehetővé a nyelvi felületeknek, hogy a *GCC* viselkedését befolyásolhassák. Minden felületnek saját ***langhooks*** struktúrát kell készítenie; Ezek a struktúrák lényegében függvénymutatókból állnak. A *GCC* közép és alsó rétege hívja meg ezeket a függvényeket ha nyelvfüggő döntetéseket kell hozni a fordítás folyamán. A ***langhooks*** szerkezetek időről időre változnak, azonban a *GCC* olyan módszerrel igényli a szerkezetek alaphelyzetbe állítását, hogy ezek a változások lényegében nem jelentenek problémát forráskód szinten. Néhány nyelvi hurok nem szabadon választható, így a felületünkhöz mindenképpen el kell készítenünk azokat. Mások ad hoc megoldások egy adott problémára. Például, a `can_use_bit_fields_p` horog kizárólag a ***gcj*** rendszer optimalizálási problémájának megoldása kedvéért került a rendszerbe.

Meghajtó készítése

Jelenleg a *GCC* megköveteli, hogy a felületünk fordításkor elérhető legyen. Nem tudunk olyan felületet készíteni amely egy már meglévő *GCC* telepítéshez csatlakozik. Ennél a lépésnél érdemes elolvasni a vonatkozó *GCC* kézikönyv oldalakat, hogy megtudjuk hogyan készítsük el a felülethez szükséges fordítási környezetet. Természetesen a legegyszerűbb módszer ha egy már meglévő nyelvi felület állományait másoljuk át és azt módosítjuk az igényeinknek megfelelően. Ezek után két fájlt készítünk el, amelyek segítenek felületünket a *GCC* meghajtóprogramba illeszteni. A ***lang-specs.h*** állomány a felületünket ismerteti a *GCC* meghajtó számára. Megmutatja a meghajtónak, hogy amikor ezt a kiterjesztést látja a parancssorban a *GCC* a mi felületünket kell meghívja. Ezen kívül az egyéb meghívandó programokkal kapcsolatos információkkal látja el a meghajtót, például kell-e assemblert futtatni a felületünk után, és hogyan adjunk át vagy módosítsunk bizonyos parancssori kapcsolókat. Ennek az állománynak elkészítése egy kis időbe telhet hiszen a leírásnak saját különös nyelve van. Más nyelvi felületek állományai sokat segíthetnek. A ***lang.opt*** a felületünkhöz tartozó parancssori kapcsolókat mutatják be. Ez az egyszerű szöveges állomány egyértelmű formátumot használ. Az egyszerű kapcsolókat például a figyelmeztető zászlókat elegendő a ***lang.opt*** állományban megadni és semmilyen további programozást nem igényelnek. a többi paramétert általunk írt nyelvi horgoknak kell kezelnie. Ezek után készítsük el a fordítás folyamatát vezérlő nyelvi horgokat. E csoport legfontosabb elemei:

- `init_options`: ez a felületünkhöz intézet első hívás a paraméterek feldolgozása előtt.
- `handle_option`: egy parancssori kapcsoló kezeléséhez meghívott horog.
- `post_options`: a parancssori feldolgozás végeztével hívódik meg. Ebben a nyelvi horogban kényelmesen meghatározhatjuk az értelmezendő állomány nevét.
- `init`: a `post_options` után hívódik meg és alaphelyzetbe állítja a felületünket.
- `finish`: A fordítás végén hívódik meg. Szükség esetén ezt használhatjuk a felület utómunkálatainak elvégzéséhez.
- `parse_file`: ez a nyelvi horog végzi el a bemeneti állományhoz tartozó teljes értelmezést, szemantikai vizsgálatot és a kódgenerálást. Lényegében itt történik a fordítási munka.

Alaphelyzetbe állítás

A *GCC* elvárja hogy nyelvi felületünk alaphelyzetbe állítsa magát. A *GCC* legtöbb része ön-előkészítő, de a különféle felületek igényeihez alkalmazkodva lehetőségünk van néhány fákkal kapcsolatos globális változó létrehozására nem típus módon. Azért azt javaslom ebbe ne nagyon mélyedjünk bele. Sokkal egyszerűbb szabványos módszerrel szabványos fa csomópontokat definiálni és saját neveket kitalálni a fákhoz amelyek mondjuk nyelvünk szabványos típusait tartalmazzák. Az alaphelyzetbe állítás során esetleg meghívhatjuk a `build_common_tree_nodes`, `set_sizetype` és `build_common_tree_nodes_2` függvényeket. A `set_sizetype` `size_t` belső megfelelőjének méretét állítja be; a legegyszerűbb ezt mindig `long_unsigned_type_node` értékre állítani. Más beállításokat is végezhetünk ebben a lépésben. A ***gcjx*** alaphelyzetbe állító kódja például különböző struktúráknak megfelelő típusokat hozunk létre amelyek a ***Java*** osztályok és metódusok leírásához szükségesek.

Fordítás GENERIC-re

A `parse_file` nyelvi horog meghívja a fordítónkat, hogy az elkészíthesse a belső adatszerkezeteinket. Feltételezve, hogy ez hiba nélkül lezajlott, a felületünk készen áll a *GENERIC* fák előállítására saját ***AST*** szerkezetünk alapján. A ***gcjx*** esetében mindez úgy történik, hogy egy különleges „vendég” ***API***-val végiglépkedünk az osztályhoz tartozó ***AST***-on. Az ***API GENERIC***-féle megvalósítása lépésenként építi fel a kód-

nak megfelelő fákat majd átadja a *GCC*-nek. A fák elkészítésének részletes ismertetése már meghaladná cikkünk kereteit. Az alábbi példák azonban bemutatják a három legnagyobb fa típust így megnézhetjük melyik hogyan is néz ki.

Type

Ez a fafajta a típusokat adja meg. Nézzünk egy példát a `gcjx`-ből a `char Java` típusra:

```
tree type_jchar = make_node (CHAR_TYPE);
TYPE_PRECISION (type_jchar) = 16;
fixup_unsigned_type (type_jchar);
```

A fák segítségével bármilyen típust le tudunk írni. Van például rekordokat, uniókat, mutatókat és különféle méretű egészeket leíró faszerkezet.

Decl

A `Decl` az értékadásnak felel meg, avagy más szavakkal egy objektumnak adott névnek. Például a forráskódban megadott helyi változó `decl` leírása:

```
tree local = build_decl (VAR_DECL, get_identifier
➥("variable_name"),
                         type_jchar);
```

A `decl`-ek különféle nevesített objektumokat jelenthetnek a programban: ***fordítási egységet (translation unit)***, függvényeket, mezőket, változókat, paramétereket, konstansokat, címkéket és típusokat. A típus `decl` a típus deklarációjának felel meg, szemben magával a típussal.

Expr

A programban előforduló különféle kifejezések leírásához sokféle *expr* fa áll rendelkezésünkre. Ezek hasonlóak a *C* kifejezésekhez, de bizonyos értelemben annál általánosabbak. A fák például nem tesznek különbséget az if utasítás és a feltételes kifejezés között - mindkettőt a `COND_EXPR` írja le, és az egyetlen különbség közöttük annyi, hogy az `if` utasítás `void` típusú. A következő kifejezés önmagát adja egy változóhoz:

```
tree addition = build2 (PLUS_EXPR, type_jchar,
local, local);
```

Az utasításoknak megfelelő fákat egy különleges közelítő *API* csatolja egymáshoz. Két utasítást (s1 és s2) a következőképpen csatolhatjuk egymáshoz:

```
tree result = alloc_stmt_list ();
tree_stmt_iterator out = tsi_start (result);
tsi_link_after (&out, s1, TSI_CONTINUE_LINKING);
tsi_link_after (&out, s2, TSI_CONTINUE_LINKING);
// most a  `result'-ba az utasítások listája került.
```

Más facsomópontok is léteznek; minden megtalálunk a *tree.def* állományban és a kézikönyvoldalon. A felületek saját fakódot is meghatározhatnak; azonban ha saját *AST*-nk van erre nem lesz szükségünk. A készülő program teljes szerkezete valószínűleg a `decl` fordítási egységre hasonlít majd, amely típusokat, változókat és függvényeket tartalmaz.

Átadás

Miután elkészítettük a függvényt, globális változót vagy típust leíró fát, amelyhez hibakeresési információt szeretnénk készíteni, a fát át kell adnunk a megfelelő függvénynek amely elvégzi a fordítás maradék részét. Jelenleg három ilyen függvény létezik: `rest_of_decl_compilation` a `decl` csomópont fordítását végzi el, a `cgraph_finalize_function` a függvények fordítását kezeli és a `rest_of_type_compilation` a típusok fordításával foglalkozik.

Hibakeresés

Bár a *GCC*-ben szép számmal találunk belső konzisztencia ellenőrzési pontokat, még így is elég könnyű összeomlást produkálni a felületünktől idegen kóddal. A legtöbb összeomlás esetén végiglépdelhetünk a vermen és megnézhetjük milyen fák voltak módosítva, amíg meg nem találjuk a hibás fakészítés okozta hibát. Ehhez a módszerhez meglepően kevés általános *GCC* ismeret is elegendő, így is hatékonyan tudunk hibákat keresni a kódban. A *GCC* rendelkezik néhány kézreálló hibakeresési lehetőséggel. A hibakeresőben meghívhatjuk a `debug_tree` függvényt, amely kinyomtatja a fát. Az `-fdump-tree` parancssori kapcsolócsaládal adott számú menet után is kiírathatjuk a fák értékét.

Tapasztalatok

A *gcjx* írása során szerzett tapasztalataim szerint az erősen típusos köztes megfelelő átalakítása fákká igen egyszerű volt. A *gcjx* fa háttérkódja, egy háttérkód a sok közül, a teljes fordító kódjának mintegy 10%-át teszi ki. Bár még befejezetlen körülbelül 6,000 kódsorból áll (nyers `wc -l` számlálás alapján) körülbelül akkora mint a bájtkód háttér. Ebből le lehet vonni azt a következtetést, hogy ha a fordítóprogramunk már készen van a *GCC*-hez kapcsolása már könnyedén megvalósítható. Tekintve, hogy a fák magas szintűek, egyetlen *RTL*-t sem néztem meg a felület készítése során és egyáltalán semmi időt nem kellett eltöltenem azzal, hogy processzor-specifikus problémákon törjem a fejemet. Amennyiben a nyelvnek amit választottál nincsenek különleges követelményei, veled is hasonló lesz a helyzet. A *gcjx* statikusan típusos *AST*-je könnyedén újrahasznosítható. Jelenleg négy háttér létezik és valószínűleg többet is írok később. Például nem lenne nehéz például olyan hátteret készíteni, amely a program kereszthivatkozásait menti adatbázisba. Vagy írhatnánk olyan hátteret amely végiglépked az *AST*-n és kikeresi a tipikus hibákat, a *FindBugs* programhoz hasonlóan. Ez az elgondolás még ösztönzőbb lehet olyan nyelvek esetében amelyek, a *Javaval* ellentétben, nem rendelkeznek gazdag analizáló készletekkel.

Jövőbeli elképzelések

A felületek írása természetesen lehetne még ennél is egyszerűbb. Például nincs szükség a *lang-specs.h* beillesztésére. Ehelyett a nyelvi felület telepíthetne egy leíróállományt, amit a *GCC* meghajtó induláskor beolvasna. Hasonlóképpen a *lang.opt* is elhagyható lenne. Egy kis munkával idővel talán az is elérhető, hogy a *GCC*-től függetlenül tudjunk nyelvi felületeket fordítani.

A cikk forrásai: ***www.linuxjournal.com/article/8138***

Linux Journal 2005. május, 133. szám

Tom Tromey