Java-fordítás GCJ-vel

Bár a Java nem túl népszerű az ingyenes projektek körében, a GCJ életképes választássá teszi.

A Java korántsem lett olyan átütő siker, mint a reklámhadjárat beharangozta, de azért népszerű nyelv, amelyet sok házon belüli és kiszolgálóoldali fejlesztésben, valamint más alkalmazásokban használnak. A szabad programok világában a Java már kisebb jelentőséggel bír, azonban így is számos projekt felhasználja. Ilyen Java-használó szabad projektre jó példa az Apache Foundation Jakarta projektje (➲ http://jakarta.apache.org), a W3C (➲ http://www.w3.org) néhány XML-eszköze és a Freenet (➲ http://freenet.sourceforge.net). De ez ügyben érdemes megnézni az FSF Java lapjait is (➲ http://www.gnu.org/software/java).

Az egyik oka annak, hogy viszonylag kevés Java-projekt készült, az ingyenes minőségi Java-megvalósítások vélt vagy valós hiánya. Pedig a Java születésétől fogva két megvalósítás is létezik. Az egyik a Kaffe (➲ http://www.kaffe.org), amit eredetileg ***Tim Wilkinson*** készített, és a részben általa alapított Transvirtual csapat fejleszt jelenleg is. A másik a GCJ (azaz a GNU Compiler for the Java language), amit 1996-ban indítottam útjára a Cygnus Solutionsnél. A GCC 3.0-s változatától kezdődően a GCJ beépített és támogatott GCC nyelv lett.

A hagyományos Java-megvalósításmodell

A Java-megvalósítás hagyományos módja két lépésből: a fordítási részből és a végrehajtási szakaszból tevődik össze. (E tekintetben a Java hasonlít a C-re.) A Java-programot a `javac` fordítja le, ami egy vagy több ***.class*** kiterjesztésű fájlt készít. Minden egyes ilyen fájl egyetlen osztály adatainak bináris megfelelője, ideértve az osztály eljárásainak kifejezéseit és utasításait is. Mindez bájtkóddá fordítódik, ami tulajdonképpen egy képzeletbeli veremalapú számítógép utasításkészlete. (Minthogy néhány lapka ténylegesen megérti a Java-bájtkód utasításkészletet, akár valós utasításkészletnek is nevezhetnénk.)

A végrehajtási szakaszt a Java Virtual Machine (JVM) kezeli, ami beolvassa és futtatja a ***.class***-fájlokat. A Sun változatát egyszerűen „Javának” hívják. A JVM-et egy Java-bájtkód utasításkészletű gépszimulátorának foghatjuk fel.

Az értelmező (szimulátor) elég sok végrehajtási többletköltséget okoz. A nagyteljesítményű JVM-ek esetében gyakori megoldás a dinamikus fordítás vagy a just in time (JIT) fordítók alkalmazása. Ebben az esetben a futásidejű rendszer észleli, ha egy eljárás olyan sokszor hívódik meg, hogy érdemes röptében elkészíteni az eljárás gépi kódját. Az eljárás további hívásai már közvetlenül ezt a gépi kódot futtatják.

A JIT hátránya az indulási többletköltség. Az eljárás lefordítása időbe telik, főként, ha némi egyszerűsítést is szeretnénk végrehajtani, és a fordítást mindig újra el kell végezni, valahányszor az alkalmazás elindul. Ha viszont csak azokat az eljárásokat akarjuk végrehajtani, amelyek a legtöbbször futnak, a többletköltséget ezek kikeresése fogja okozni. További gond, hogy egy jó JIT összetett és meglehetősen nagy helyet foglal el (ezenkívül az elkészített kódnak is szüksége van helyre, ami az eredeti bájtkód által használt helyen felül további helyigényt jelent). E területek kis része lehet az osztott memóriában.

A hagyományos Java-megvalósítási módszerek ezen felül nem működnek együtt túl jól más nyelvekkel. Az alkalmazásokat más módszerrel kell telepíteni (Java Archive, azaz ***.jar***-fájlokkal, és nem végrehajtható állományokkal); nagy futásidejű rendszert igényelnek, ráadásul a Java és a C/C++ közötti átjárás lassú és kényelmetlen.

A GCJ-megoldás: előrefordítás (Ahead-of-Time Compilation)

A GCJ-projekt megközelítése gyökeresen eltérő. A Javát egyszerűen egy új programozási nyelvnek tekintjük, és pontosan úgy járunk el, ahogy a többi fordításalapú nyelv esetén. Mivel a Cygnus régóta foglalkozott a GCC-vel, ami már eleve számos különféle nyelv fordítására volt képes (C, C++, Pascal, Ada, Modula2, Fortran, Chill), elég logikusnak tűnt, hogy a Javát is megpróbáljuk GCC-vel helyi kódra fordítani.

Egészében véve a Java-program lefordítása sokkal egyszerűbb, mint egy C++-program fordítása, mivel a Java nem tartalmaz se sablonokat, se előfeldolgozót. A típusrendszer, az objektummodell és a kivételkezelő modell ugyancsak egyszerűbb. A Java lefordításához a programot alapvetően egy elvont szintaktikai faként képzeljük el, ahol ugyanazokat az adatszerkezeteket használjuk, amelyeket a GCC minden egyéb nyelvhez is használ. Minden egyes Java-szerkezethez a C++ belső megfelelőjét használjuk, a többit már a GCC elvégzi.

A GCJ ezek után már élvezheti a GNU-eszközökbe épített összes egyszerűsítés és bővítmény előnyét. Ilyen egyszerűsítésre példa az általános alkifejezés-kiiktatás (sub-expression elimination), erőcsökkentés (strength reduction), ciklusegyszerűsítés (loop optimization) és a regiszterfoglalás (register allocation). Ezenkívül a GCJ kifinomultabb és időigényesebb javítást is végrehajthat, mint ami a röptében fordítók (just in time) esetében megengedhető. Akadnak olyanok is, akik vitatják ezt, mondván, a JIT jobban alkalmazkodó és testreszabottabb egyszerűsítésre is képes lehet (például a kódot a pillanatnyi végrehajtás függvényében megváltoztathatja). Meg kell hagyni, a Sun HotSpot módszere, ami ezeken az elveken alapul, igencsak lenyűgöző munkát végez. Az igazat megvallva a GCJ-vel fordított program futása valóban nem mindig lesz láthatóan gyorsabb, mintha egy JIT-alapú Java-megvalósításban futtatnánk; sőt néha akár lassabb is lehet, de ez sokkal inkább annak tudható be, hogy nem maradt időnk elkészíteni a javás egyszerűsítéseket és finomhangolni a GCJ-t. A GCJ gyakran lényegesen gyorsabb, mint egyéb JVM-ek, és egyre gyorsabb lesz, ahogy az emberek fejlesztik.

A GCJ nagy előnye az indulási sebesség és a mérsékelt memóriahasználat. Eredetileg úgy tartották, hogy a bájtkód sokkal helyhatékonyabb lesz, mint a helyi saját utasításkészlet. Bizonyos tekintetben ez valóban így is van, de nem árt, ha belegondolunk, hogy a ***.class***-fájloknak körülbelül a felét szimbolikus (nem utasítás-) adatok foglalják el. Ezek a szimbólumok minden egyes ***.class***-fájlban ismétlődnek, ugyanakkor az ELF végrehajtható állományok vagy programkönyvtárak ennél sokkal jobb közös használatot tesznek lehetővé. Ugyanakkor a bájtkód

igazán hátrányba kerül a helyi kóddal szemben a JIT-et alkalmazó JVM-ben felhasznált memória tekintetében. Ha JIT-fordítással elindítjuk a Sun JVM-jét, az alkalmazásosztályok hatalmas időmennyiséget és memóriát fognak fogyasztani, például a Sun IDE Forte for Java (ami NetBeans néven nyílt forrású változatban is elérhető) tényleg gigászi lesz. A NetBeans indítása 74 MB memóriát fogyaszt (a `top` parancs jelentése szerint), még mielőtt egyáltalán bármivel elkezdtünk volna dolgozni. A Java-alkalmazások által elfoglalt memóriaterület megnehezíti a telepítésüket is. Jól mutatja ezt az általam készített (nem túl aktív) Jemacs projekt (Jemacs.sourceforge.net), ami az Emacset Swing segítségével Java alatt próbálja meg létrehozni (no meg az alább tárgyalt Kawa felhasználásával az Emacs Lisp támogatásához). A Sun JDK1.3.1-ben indított egyszerű szerkesztőablak 26 MB-ot fogyaszt (a `top` szerint). Ezzel szemben az Xemacs mindössze 8 MB-ot eszik.

A Kawa-próbakészletet GCJ-vel és JDK1.3.1-el futtatva a GCJ nagyjából kétszer olyan gyorsnak bizonyult, nagyjából fele annyi laphibát okozott (a `time` parancs szerint), és körülbelül 25 százalékkal kevesebb memóriát fogyasztott (a `top` alapján). A próbakészlet-parancsfájl, amely a Java-környezetet többször elindítja, túlságosan sok különböző dolgot futtat, így a JIT nem tud érvényesülni (ez a JDK-ra nézve hátrányos). Ugyanakkor betölt és interaktívan futtat Scheme-kódot, így a GCJ-nek az értelmezőjén keresztül kell lefuttatnia (ami viszont a GCJ-re nézve hátrányos). Ez a kísérlet nem számíthat igazi teljesítménymérésnek, de azt azért megmutatja, hogy a GCJ-től már jelen állapotában is teljesítménynövekedést várhatunk. (Mint általában, ha a teljesítményre vagyunk kíváncsiak, futtassuk le a várható munkafajták keverékén alapuló saját teljesítménymérő tesztünket.)

A GCJ egyéb előnyökkel is rendelkezik: hibakeresés GDB-vel és csatolófelület a C/C++-hoz (később még lesz szó róla). Végül a GCJ az ipari szabványnak számító GCC-n alapuló ingyenes program, ami lehetővé teszi, hogy szabadon módosítsuk, átültessük vagy terjesszük.

Néhányan kifogásolni szokták, hogy az előfordítással (ahead-of-time compilation) elveszik a bájtkód „írjuk meg egyszer – futtassuk akárhol" hordozhatósága. Csakhogy a kötekedők figyelmen kívül hagyják azt a tényt, hogy a terjesztés nem azonos a telepítéssel. Nem azt javasoljuk, hogy a helyi kód legyen a terjesztési formátum, hacsak nem valamilyen egyedi kiépítéshez készült előregyártott csomagról van szó (pl.: RPM-ről). A Java-bájtkódot továbbra is használhatjuk terjesztési formátumként, még ha nincs is semmilyen komoly előnye a Java-forráskóddal szemben. (A Java-forráskód kevesebb együttműködési nehézséggel küzd, mint a C- vagy a C++-források.) Azt javasoljuk, hogy amikor Java-alkalmazást telepítünk, fordítsuk le a helyi kódra, ha esetleg ne lenne már eleve lefordítva.

Java-program fordítása GCJ-vel

A GCC-vel végzett Java-fordítás ismerősnek fog tűnni mindenkinek, aki már fordított C- vagy C++-programot GCC-vel. A *MyJavaProg.java* Java-program fordításához a következőket gépeljük be:

```
gcj -c -g -O MyJavaProg.java
```

Összeépítéshez használjuk az alábbi parancsot:

```
gcj --main=MyJavaProg -o MyJavaProg
➥MyJavaProg.o
```

Éppen olyan, mintha *mycxxprog.cc* C++-programot fordítanánk le:

```
g++ -c -g -O mycxxprog.cc
```

majd összeépítenénk, hogy létrehozzuk a `mycxxprog` végrehajtható állományt:

```
g++ -o mycxxprog mycxxprog.o
```

Az egyetlen új dolog a `-main=MyJavaProg` lehetőség. Erre azért van szükség, mert a Java-osztályokat gyakran `main` tagfüggvénnyel látjuk el, amit aztán kipróbálásra, esetleg önálló kis eszközként használunk. Ezért amikor egy csoport Java-fordított osztályt fordítunk egybe, több `main` tagfüggvény is lehet, az összeépítőnek (linker) meg kell mondanunk, hogy melyiket kell meghívni közülük, amikor az alkalmazás elindul. Lehetőségünk nyílik arra is, hogy Java-osztályok halmazát osztott programkönyvtárba (*.so*-fájlba) fordítsuk. Valójában a GCJ futásidejű rendszer is ilyen *.so*-fájlba fordul. Bár ennek részletei már egy másik cikk témájába tartoznának, akit érdekel, nézzen körül az alább tárgyalandó Kawa `Makefile`-jában, hogy láthassa, miként is működik.

A GCJ képességei és korlátai

A GCJ nem csupán fordítóprogram. A Sun JDK-hoz hasonló teljes körű Java-környezetnek szánták. Ha megadjuk a `-C` kapcsolót, a `gcj` hagyományos *.class*-fájlokat fordít. A cél az volt, hogy a `gcj -C` parancsot a sunos `javac` parancs kiváltására is fel lehessen használni.

A GCJ-ben megtalálhatjuk a bájtkódértelmezőt (***Kresten Krab Thorup*** jóvoltából) és a teljes értékű `ClassLoadert`. Az önálló `gij` program a Sun Java-programjának kiváltására használható. A GCJ a GCC 3.0-ban megjelent `libgcj`-t használja. A futásidejű könyvtárban megtalálhatjuk a futásidejű támogatás magvát, ***Hans Boehm*** figyelemre méltó hagyományos szemétgyűjtőjét, a bájtkódértelmezőt, illetve az osztályok jókora könyvtárát. Jogi és technikai szempontok miatt a GCJ nem tartalmazhatja a Sun osztálykönyvtárát, ezért saját változattal bír. A GNU Classpath Projekt ugyanazt az engedélyt és FSF jogvédelmét alkalmazza, mint amit a `libgcj` és `libstdc++` használ, így a két projekt osztályai egyesíthetők. Mi a GPL-t használjuk, azzal a különleges kitétellel, hogyha végrehajtható állományt a `libgcj`-t más fájlokkal egybefordítva készítünk, az önmagában még nem feltétlenül jelenti azt, hogy az eredménynek is GPL alá kell tartoznia. Így akár üzleti programok is készíthetők a szabványos C++ vagy Java futásidejű könyvtárakkal.

A `libgcj` könyvtár a GUI nélküli alkalmazásokhoz szükséges legtöbb szabványos osztályt tartalmazza, ide értve a ***java.lang***, ***java.io***, ***java.util***, ***java.net***, ***java.security***, ***java.sql*** és a ***java.math*** osztályok csomagjait. A legfontosabb hiányzó osztályok a grafika készítését támogató AWT- vagy Swing-bővítmények. A legtöbb magas szintű AWT osztály készen van, de az alacsony szintű osztályok még nincsenek olyan fokon, hogy érdemben használni lehessen őket. Várjuk a segítő szándékú jelentkezőket!

Együttműködés a C/C++-rendszerekkel

Bár Javában sok mindent meg tudunk csinálni, néha mégis szükség lehet rá, hogy más nyelvekben írt programkönyvtárakat hívjunk meg. Ennek az egyik oka az lehet, ha olyan alacsony szintű kódot akarunk használni, amit a Javában nem lehet elkészíteni, de ki akarjuk használni egy olyan könyvtár lehetőségeit, amit nem szeretnénk újraírni, vagy sebességre menő alacsonyszintű kódolást végzünk. Mindezt megtehetjük egyszerűen úgy, hogy őshonosnak (native) jelölünk meg néhány Java-tagfüggvényt, majd a tagfüggvény testének létrehozása helyett a megvalósítást

valamilyen más nyelven készítjük el. A Sun 1997-ben kiadta a Java Native Interface (JNI) csatolófelületét, ami a C vagy C++ nyelveken készíthető őshonos tagfüggvények készítését szabályozó szabvány. A JNI legfőbb célja az átültethetőség, azazhogy az egyik Java-megvalósításhoz készült őshonos tagfüggvény a másik Java-megvalósításon újrafordítás nélkül futtatható legyen. Ez azonban a zárt forrású, terjesztett bináris világ céljaihoz készült, és jóval kevesebb haszna van a szabad programok világában, legfőképpen azért, mert újra kell fordítanunk, ha processzort vagy operációs rendszert váltunk.
A megfelelőség (compatibility) érdekében JNI alatt mindent közvetett módon, függvénytáblákon keresztül végzünk el. Ez azonban igencsak lelassítja a JNI-t. Még ennél is kellemetlenebb, hogy az összes ilyen függvényt fárasztó és hibákra hajlamos szabályok alapján kell létrehoznunk. Bár a GCJ támogatja a JNI-t, választási lehetőséget is nyújt. A Compiled Native Interface (CNI, amelyet egyben Cygnus Native Interface-ként is olvashatunk) azon az elképzelésen alapul, miszerint a Java alapvetően a C++ alhalmaza, és a GCC ugyanazt a hívási módszert alkalmazza mind a C++, mind a Java esetében. Így természetesen lehetséges C++ nyelvet használó Java-tagfüggvényt írni, és a C++ szabályos írásmódját felhasználva érhetjük el a Java-mezőket, hívhatjuk meg a Java tagfüggvényeket. Mivel ugyanazt a hívási elgondolást és adatkiosztást alkalmazzák, a C++ és a Java között nincsen szükség átalakításra vagy mágikus bűvészkedésre.
A CNI- és JNI-példák bemutatása már egy következő cikk témája lesz. A GCJ kézikönyv elég jól ismerteti a CNI-t, (➲ http://gcc.gnu.org/onlinedocs/gcj) a libgcj forrásokban pedig számos példát találhatunk.

Kawa: sémák fordítása Javára

A Java-bájtkód a Java-programok eléggé közvetlen kódolása, amit nemigen terveztek bármi másra. Ennek ellenére használják más nyelven írt programok kódolására is.
A ➲ http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html címen olvasható listában megnézhetjük hogy, milyen más programozási nyelveket készítettek a Java fölé. A legtöbb ezek közül értelmező, de néhány közülük ténylegesen bájtkódot készít. Az első egy az egyben felhasználhatja a GCJ-t; az utóbbi esetleg a GCJ-t használhatja a natív kód fordítására.
Az egyik ilyen fordító a Kawa, amit 1996 óta fejlesztek. A Kawa a Javát használó nyelvek megvalósítására használható eszközkészlet és egyúttal a Scheme programozási nyelv megvalósítása. A GCJ segítségével minden fizetős program nélkül elkészíthetjük a Kawát. A Kawa honlapon (➲ http://www.gnu.org/software/kawa) megtaláljuk a GCJ és Kawa letöltésére és telepítésére vonatkozó utasításokat.
A Kawa felhasználói beavatkozást igénylő (interactive) módban is használható. Most megadunk egy faktoriális függvényt, majd meghívjuk:

```
$ kawa
#|kawa:1|# (define (factorial x)
#|(---:2|#   (if (< x 2) x (* x (factorial
➥(- x 1)))))
#|kawa:3|# (factorial 30)
265252859812191058636308480000000
```

A dolognak az az érdekessége, hogy a faktoriális függvényt a Kawa bájtkóddá fordította, és mint új osztályt azonnal be is töltötte. A folyamat a Java `ClassLoader` szerkezetét használja az új osztály futásidejű meghatározásához, létrehozva az osztály bájtkódját tartalmazó bájttömböt. Az új osztály eljárásai a GCJ bájtkódértelmezőjén keresztül futnak le.
Természetesen a kódot általában kényelmesebb fájlba helyezni:

```
$ cat > factorial.scm
(define (factorial x)
(if (< x 2) x (* x (factorial (- x 1)))))
(format #t "Factorial ~d is ~d.~%~!" 30
➥(factorial 30))
^D
$ kawa -f factorial.scm
Factorial 30 is
265252859812191058636308480000000.
```

Megnövelhetjük a Scheme-kód teljesítményét, ha a Kawa segítségével előrefordítjuk, egy vagy több ***.class***-fájlt hozva létre:

```
$ kawa --main -C factorial.scm
(compiling factorial.scm)
```

A lefordított fájlt aztán betölthetjük:

```
$ kawa -f factorial.class
Factorial 30 is
➥265252859812191058636308480000000.
```

Az osztályfájl helyi kódra történő fordításához a `gckawa`-t használhatjuk, ami nem más, mint a szükséges környezeti változókat (`LD_LIBRARY_PATH` és `CLASSPATH`) beállító, majd a GCJ-t meghívó parancsfájl:

```
$ gckawa -o factorial --main=factorial
➥-g -O factorial*.class
```

A ***factorial*.class*** névben használt helyettesítő (joker) karakter használata itt szükségtelen, de hasznos lehet, ha a Kawa több ***.class***-fájlt készít.
Ezt követően végrehajthatjuk az eredményül kapott normális GNU/Linux ELF végrehajtható állományt. A `libgcj.so` (GCJ futásidejű könyvtár) és a `libkawa.so` (a Kawa futásidejű könyvtár) kapcsolódik hozzá.
Más nyelvekhez is használható ugyanez a megközelítés. Jelenleg a W3C új XML-lekérdező nyelvének, az Xquery-nek megvalósításán dolgozom a Kawa segítségével. A GCJ-vel készült alkalmazások közé tartozik néhány Apache-modul, a GNU-Paperclips és a Jigsaw.

Összegzés

A GCJ fejlesztése mostanában igen nagy pezsgést mutat, és számos feladathoz megbízható alapot jelenthet. Reméljük, hogy sikerült kedvet csinálni a Javának a szabad programjainkban történő felhasználásához, a GCJ-hez mint előnyben részesített Java-megvalósításhoz, illetve néhányan talán segítenek majd még jobbá tenni a GCJ-t.

Linux Journal 2003. január, 105. szám

Per Bothner
Az 1980-as évektől kezdve GNU és programon dolgozik. A Cygnusnál a GCJ projekt műszaki vezetője volt. Jelenleg független szakértő.
A ➲ http://www.bothner.com/per címen érhető el.