Compilerbau für die Common Language Runtime

Aufbau des GCC

Quellorganistation: Probleme

- Modularisierung des Compilers:
  - verschiedene Übersetzungsphasen (Präprozessor, Compiler, Assembler, Linker)
  - verschiedene Frontends und Backends
  - Compiler-Compiler: Programme, die Quelltext des Compilers generieren
  - Comilerabhängige Laufzeitbibliotheken
- Portabilität, Cross-Compilerbau:
  - Buildsystem: System, auf dem der Compiler übersetzt wird
  - Hostsystem: System, auf dem der Compiler läuft
  - Targetsystem: System, für das der Compiler Code erzeugt
  - gleichzeitige Installation von Compilern für verschiedene Targets

Modularisierung: Toplevel-Verzeichnis

- Cygnus configure: Ein Build-Lauf soll gesamte Werkzeugkette für Host/Target-Kombination erzeugen können
  - *Overlay*-Struktur: gcc, binutils (as, ld), gdb, make(?) können alle in das gleiche Verzeichnis integriert werden, und “am Stück” übersetzt werden
- gcc: enthält Compiler, Laufzeitbibliotheken, Supporttools für Compiler
  - gcc: Verzeichnis für GCC selbst
  - fixincludes:Übersetzer zur Anpassung von Headerfiles
  - boehm-gc, libada, libffi, libgfortran, libjava, libmudflap, ›libobjc, **libruntime,** libssp, libstdc++v3: Laufzeitbibliotheken
  - libcpp, libiberty, libintl: vom Compiler selbst verwendete Bibliotheken

gcc

- Verzeichnis gcc selbst: Buildprozess, C compiler, Definition von Compiler-Datenstrukturen (tree, RTL), *middle end*
- config: Backends
  - je ein Unterverzeichnis pro Prozessor
- doc: texinfo-Dokumentation
- ginclude: C-Headers für *free-standing implementation*
- po: *message catalogs* für Compilermeldungen
- testsuite:
- **cil**, cp, fortran, java, objc, objcp, treelang: front ends

Übersetzung

- GNU autoconf (+Cygnus configure?)
- Aufruf von configure im Toplevel-Verzeichnis ruft rekursiv benötigte weitere configure-Skripts auf
  - Toplevel-configure benötigt alle Parameter von allen gerufenen configures
- Übersetzung i.d.R. in separatem Dateibaum:
  - mkdir h8300
  - cd h8300
  - ../configure --target=h8300-hms
- *full bootstrap*: “make bootstrap”
  - 3 Phasen: Phase 1 mit host cc, Phase 2 mit Phase-1-gcc, Phase 3 mit Phase-2-gcc (danach Vergleich von Phase 2 und Phase 3)
- *quick*: “make”

configure-Optionen

- --with und --enable (--without/--disable)
- hier nur ausgewählte Optionen von gcc/configure (volle Liste jeweils mit --help)
- --build, --host, --target
  - Default für build: geraten durch config.guess
  - Default für host: wie --build
  - Default für target: wie --host
- --enable-languages=s1,s2
  - z.B. --enable-languages=c,cil
  - Default abhängig von <lang>/config-lang.in:build_by_default
- --enable-checking: Selbstdiagnose des Compilers
  - konsistente Verwendung von polymorphen Typen (tree, rtl), Speicherverwaltung (gc), ...

Compilerpässe

- Treiberprogramm (gcc, g++, gjc, ...)
  - Implementiert in gcc/gcc.c
  - Verarbeitung des “spec”-Files (Beschreibung der Pässe)
  - gcc -dumpspecs
  - erweiterbar durch <frontend>/lang-specs.h
- *1 (cc1, cc1obj, cc1plus, jc1, **cil1**, tree1): “erster” Pass
  - C/C++: eventuell vorher Präprozessor (cpp)
    - seit gcc 3.x: integrierter Präprozessor
  - jeder Compiler wird mit libbackend.a verbunden
  - Quelldateien des Passes in <frontend>/Make-lang.in
- Weitere Pässe (as, collect2, ld, ...) nicht Teil von gcc

Target Triplets

- Eindeutige Benennung von Plattformen durch autoconf
  - <arch>-<vendor>-<system> (optional: -<env>)
  - u.U. Abweichung von kanonischer Form
  - z.B. sparc-sun-solaris10, powerpc-apple-darwin8, i486-linux
- Treiber installiert als:
  - gcc (g++, ...)
  - <triplet>-gcc (etwa: powerpc-apple-darwin8-gcc)
    - analog: powerpc-apple-darwin8-as
  - /usr/local/powerpc-apple-darwin8/bin/gcc (genauso: as, ld)
- Pässe installi0ert unter /usr/local/lib/gcc-lib/<triplet>/<version>

Frontend-Struktur

- gcc/toplev.c: Eintrittspunkt (main)
- build process: Make-lang.in, config-lang.in
- hooks: deklariert in gcc/lang-hooks, definiert durch globale Variable lang_hooks
  - Frontend-Name
  - Initialisierung des Frontends, Verarbeitung von Kommandozeilenoptionen (INIT, INIT_OPTIONS, HANDLE_OPTIONS, ...)
  - Verarbeitung einer Eingabedatei (PARSE_FILE)
  - Umsetzung spezifischer AST-Konzepte in universelle (EXPAND_EXPR, EXPAND_DECL)
  - Diagnose, debugging (PRINT_DECL, PRINT_TYPE, ...)
  - ...
  - cil: definiert in cil-decl.c

Abstrakte Syntaxbäume

- Polymorpher Datentyp tree (aus tree.h): union tree_node*
  - Jede union-Variante beginnt mit struct tree_common
  - Varianten: common, int_cst, real_cst, identifier, var_decl, const_decl, type, exp, ...
  - tree_common: Typinformation in Feld code
    - außerdem: Verkettung (chain), Typ, ...
- Zugriff auf Felder von tree nur über Zugriffsmakros
  - enable-checking: Makros überprüfen typrichtigen Zugriff
- TREE_CODE(x): Art des Knotens
  - Liste der möglichen Knotentypen in tree.def
  - erweiterbar durch Frontend
- TREE_CODE_CLASS(x): Gruppierung der Codes in “abstrakte Basisklassen”

Treecodeklassen

- tcc_type: Datentypen (int, float, void*, ...)
- tcc_declaration: Deklarationen (Funktionen, Variablen, Typdeklarationen)
- tcc_unary: Unäre Operatoren
- tcc_binary: Binäre arithmetische Operatoren
- tcc_comparison: Vergleichsoperatoren
- tcc_statement: Anweisungsausdrücke
- tcc_expression: Sonstige Ausdrücke
- ...

Zugriffsmakros

- Tests (Prädikate): Endung _P
  - TYPE_P, DECL_P, ..., TREE_READONLY_P
- Zugriffsmakros für alle Knotentypen: TREE_TYPE, TREE_CHAIN
- Zugriffsmakros für bestimmte Klassen oder tree codes: DECL_NAME, DECL_SOURCE_FILE, DECL_SOURCE_LINE, TYPE_SIZE, TYPE_FIELDS,

Speicherverwaltung

- Anlegen von Knoten durch Funktion build
  - build(code, type, ...) /* Zahl der Parameter code-abhängig, siehe tree.def */
  - GCC 4: typsicherer Versionen build0, build1, ...
- knotenspezifische Konstruktoren: build_int_cst, build_real, ...
- Speicherfreigabe automatisch über garbage collection
  - Makromaschinerie zum automatischen Verfolgen von Zeigern
- Bezeichner: Singletons (unique)
  - Allozierung/Auffindung über get_identifier
  - Attributierung von Bezeichnern: Symboltabelle

Listen/Arrays

- Jeder tree_node kann automatisch in einer Liste sein: TREE_CHAIN
- Außerdem: Knotentyp TREE_LIST
  - Verkettung über TREE_CHAIN von TREE_LIST-Knoten
  - **zwei** gespeicherte Werte: TREE_PURPOSE, TREE_VALUE
    - falls nur ein Wert verkettet werden muss, ist TREE_PURPOSE NULL
  - Allozierung über tree_cons
  - Verkettung über chainon
  - Zugriff über Makros, tree_last, list_length, ...
- Vektoren: Knotentyp TREE_VEC
  - effizienter indizierter Zugriff

TREE_CODEs

- definiert in tree.def
- ERROR_MARK: im Eingabeprogramm wurde ein Fehler erkannt; ERROR_MARK dient zur Fehlerstabilisierung
  - Singleton error_mark_node
- IDENTIFIER_NODE: Bezeichner
- ENUMERAL_TYPE, INTEGER_TYPE, REAL_TYPE, POINTER_TYPE, RECORD_TYPE, ...: Typen
  - Singletons: integer_type_node, pointer_type_node, ...
- FUNCTION_DECL, VAR_DECL, FIELD_DECL: Deklarationen
  - VAR_DECL auch als Ausdruck verwendbar

TREE_CODEs für Ausdrücke

- COMPONENT_REF, INDIRECT_REF: Speicherzugriff
- MODIFY_EXPR: Zuweisung
- COMPOUND_EXPR: sequentielle Berechnung
- COND_EXPR: Bedingte Berechnung
- CALL_EXPR: Funktionsruf
- WITH_CLEANUP_POINT_EXPR, CLEANUP_POINT_EXPR, TRY_CATCH_EXPR: Ausnahmebehandlung
- PLUS_EXPR, MULT_EXPR, TRUNC_DIV_EXPR, ...: arithmetische Ausdrücke
- LT_EXPR, GT_EXPR, ...: relationale Ausdrücke
- CONVERT_EXPR: Typkonvertierung

Kontrollfluss-Ausdrücke

- LABEL_EXPR, GOTO_EXPR: Sprünge
- RETURN_EXPR
- LOOP_EXPR
- SWITCH_EXPR
- STATEMENT_LIST: Sequentielle Ausführung
  - Verwendung zusammen mit

Ablauf der Übersetzung

- Übertragung von tree nodes in RTL (Register-Transfer-Language)
  - Abstrakte Maschine mit primitiven sequentiellen Operationen
  - beliebig viele Register
  - Danach: Übertragung von RTL in Maschinencode des Zielsystems
- expression-at-a-time: Aufbau von tree nodes für einen einzelnen Ausdruck, danach Überführung von trees in RTL
  - Kontrollfluss nur auf RTL-Ebene repräsentiert
  - in GCC 4 nicht mehr unterstützt
  - *expression folding* auf tree-Repräsentation
- function-at-a-time: Aufbau von tree nodes für gesamte Funktion
  - Optimierungspässe auf tree-Repräsentation (z.B. tree SSA)
- translation-unit-at-a-time: Aufbau von tree nodes
  - Optimierung auf Dateiebene mittels *tree nodes* (z.B. Inter-Procedure-Optimization, IPO)

Ablauf der Übersetzung (2)

- Allozierung von Datenstrukturen i.d.R. für jede Funktion
  - lokale Variablen
  - RTL
  - tree nodes für den Funktionskörper (function-at-a-time)
- Globale Variablen zur Speicherung des aktuellen Kontexts
  - current_function_decl: DECL_-node für aktuelle Funktion
  - cil1: make_current_function_decl, compile_function
- langlebige Datenstrukturen: Typdefinitionen, Deklarationen von global Variablen und Funktionen
  - cil1: Iteration über alle Typen zur Deklation (parse_classes), Iteration über alle Methoden zur Übersetzung (parse_all_methods), Ausgabe des Codes (unit-at-a-time)

Weitere Aspekte

- Aufbau der RTL
- Aufbau der Maschinenbeschreibungen, Generierung von Assemblercode