

Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

29. November 2005

Der Plan für Heute

- Bisher: Übersicht über Grundideen
 - Gruppen von Konzepten
 - Beziehungen zwischen Konzepten
 - Grundlegende Entscheidungen
 - Relativ wenige Details
- Jetzt: Beitrag der einzelnen Module zur Übersetzung
 - Vorgehen durch den Compiler
 - Sprachkonstrukte getrennt betrachten
 - Details ausprogrammieren

Aktueller Stand Übersetzerprojekt

- Nach Übungsblättern
 - AAST im Griff (Blatt 6)
 - C-AST im Griff (Blatt 6)
 - Vorstellung vom Übersetzungsprozess (Blatt 7)
- Erster Versuch "Java Programme in C"
 - Datenstrukturen für einfache Objekte
 - Methoden werden Funktionen mit `this`-Parameter
 - Methodenaufruf übergibt das `this`-Objekt
- Noch nicht behandelt
 - Objekte enthalten Methoden (Polymorphie)
 - Namenskonflikte zwischen Klassen
 - Typen

Plan: Kurzform

→ Wir müssen die Felder dieser Tabelle systematisch spaltenweise ausfüllen

	Klassen	Objekte	...	Methoden	Methodenaufruf
Syntax	✓	✓		✓	✓
AAST	✓	✓		✓	✓
Rewrite					
Tc					
Layout					
Cg					

Übersicht Compiler

- Aufgaben der einzelnen Module schon zusammengefaßt
- Jedes Modul behandelt einen bestimmten Aspekt der Übersetzung
- Innerhalb jedes Moduls M gilt die Konvention
Wenn m die Aufgabe des Moduls M ist und t ein Typ aus der AAST, dann erfüllt die Funktion m_t die Aufgabe von M für den Typ m .
- Wenn t ein algebraischer Datentyp ist, besteht m_t üblicherweise aus einer Fallunterscheidung, eventuell mit Rekursion.
- Der Einstiegspunkt ist `m_tydecls`
- Hinweis: `make interfaces` erzeugt die `.mli` Dateien mit Typen

Klassen und Objekte im Detail

- Erinnerung
 - Objekte sind Instanzen von Klassen
 - Instanzen werden C-Structs mit den Instanzvariablen
 - Objekte haben Referenzsemantik → C-Zeiger
 - Instanziierung
 - Speicheranforderung
 - Konstruktoraufruf
 - Methodenaufruf: Zielobjekt als `this`-Parameter übergeben
- Unklar: Methoden im Objekt selbst speichern
- Ziel: Erweiterung mit Vererbung

Übersicht Compiler

- Beispiel für die Konvention: Typchecker Tc
- Aufgabe: `chk`: überprüft Typregeln für AAST-Knoten
→ Wirft Exception bei Fehlern

```
val chk_tydecls : Annot.elem list -> unit
val chk_tydecl  : Annot.elem      -> unit
val chk_cls     : Annot.cls       -> unit
val chk_ctor    : Annot.ctor      -> unit
val chk_meth    : Annot.meth      -> unit
val chk_sm      : Annot.sm        -> unit
val chk_exp     : Annot.exp       -> Annot.ty
```
- Abstieg in den Annotierten Abstrakten Syntax Baum
- Fragen an jedem Knoten: Was muß das Modul leisten für ...
 - ... für die Kinder des aktuellen Knotens
 - ... in den möglichen Fällen des aktuellen Knotens

Polymorphie

```
class A {
  int i;
  void f() { System.out.println("A.f"); }
}
class B extends A {
  void f() { System.out.println("B.f"); }
}
class Main {
  static void callF(A a) { a.f(); }
  public static void main(String argv[]) {
    callF(new A());
    callF(new B());
  }
}
```

- B erbt von A und überschreibt die Methode `f`
- B-Instanzen führen bei Aufruf von `f` den Code aus Klasse B aus
- Objekte enthalten in irgendeiner Form ihre Methoden

Idee: Funktionszeiger in Objekten

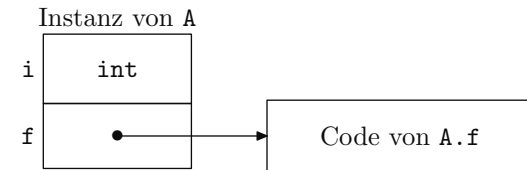
- C gibt uns Zeiger auf Funktionen
- Jede Instanz enthält Zeiger auf ihre eigenen Methoden
- Initialisiere Zeiger bei Instanziierung mit den Methoden einer Klasse
- Methodenaufruf entnimmt aufzurufende Methode aus dem Objekt

Auswertung

- ✓ Beim Methodenaufruf enthält Zielobjekt eigene Methoden
- ✓ Polymorphie beim Vererbung bleibt möglich
- ✓ Instanziierung entscheidet über Methoden des Objektes
- ✗ Speicherbedarf: Ein neues Feld pro Methode des Objektes

Funktionszeiger in Objekten

- Laufzeitdatenstruktur für Instanzen



- Instanziierung `new A()` initialisiert Funktionszeiger

`(t=(...)malloc(...), t->f = A_f, ctor(t), t)`

- Methodenaufruf `a.f()` nimmt Funktionszeiger aus Objekt

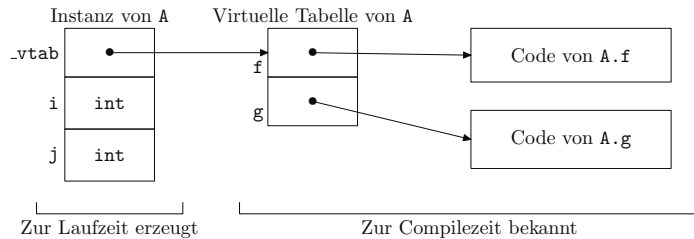
`(t=a, t->f(t))`

Virtuelle Tabellen

- Ansatz aus C++: Auslagern der Methodenzeiger in separate Tabelle
- Diese [virtuelle Tabelle](#)
 - wird von Instanzen mit einem einzigen Zeiger referenziert
 - ist nur einmal pro Klasse im Speicher enthalten
 - wird beim Programmstart initialisiert
- Methodenaufruf
 - Berechnet Zielobjekt
 - Greife auf virtuelle Tabelle des Zielobjektes zu
 - Lese gewünschten Methodenzeiger
 - Rufe Methode auf

Virtuelle Tabellen

```
class A {
  int i;
  int j;
  void f() { ... }
  void g() { ... }
}
```



Virtuelle Tabellen: Neue Aufgaben

- Eindeutige Namen finden
 - Instanz-Structs
 - Virtuelle Tabelle
 - Methoden
- Virtuelle Tabelle in im Compiler darstellen
 - Modul `Rt_tables` (für *run-time tables*)
 ⇒ Compilezeitumgebung als statische Repräsentation der Laufzeitumgebung
- Virtuelle Tabelle initialisieren
- Instanziierung erweitern: `_vtab` muß initialisiert werden

Übersicht: Klassen im Compiler

Syntax	
Annot	Prüfung: Alle Typen definiert? ✓ Nachladen von referenzierten Klassen ✓
Rewrite	(Später)
Tc	Methoden- und Konstruktorrümpfe auf Typkorrektheit prüfen
Layout	Namen für C-Funktionen & -structs Aufbau von Instanz-Structs & virtuellen Tabellen
Cg	Beschreibung von Structs in C-Deklarationen umsetzen Methoden- und Konstruktor-Rümpfe in C-Anweisungen übersetzen

Virtuelle Tabellen in `Rt_tables`

```
type vtab = {
  mutable vtab_decl_nm : string;
  mutable vtab_def_nm   : string;
  mutable vtab_entries : vtab_entry list;
}
```

- `vtab` beschreibt virtuelle Tabelle einer Klasse
 - `vtab_decl_nm`: Struct-Name auf C-Ebene
 - `vtab_def_nm`: Die entsprechende globale Variable mit Funktionszeigern
 - `vtab_entries`: Der interne Aufbau der Tabelle (→ nächste Folie)
- (Unterscheide Deklaration und Definition wie in C)

Methoden in Virtuellen Tabellen

```
type vtab_entry = Vtab_meth of vtab_meth
type vtab_meth = {
  vtm_decl : Annot.meth;
  mutable vtm_def : Annot.meth;
}
```

- Bisher einziger Inhalt von Tabellen: Methoden
- Pro Methode vtab_meth
 - Die Deklaration der Methode
 - Bestimmt Position in virtueller Tabelle
 - Die Definition der Methode, aus der der Code genommen wird
 - Ergibt konkreten Funktionszeiger in virtueller Tabelle
- Erweiterung für Überschreiben bleibt möglich:
Unterschiedliche Einträge in vtm_def für gleichen vtm_decl Eintrag

Zwischenstand

- ✓ Laufzeit-Datenstruktur für Objekte → Virtuelle Tabellen
- ✓ Compilezeit-Repräsentation dieser Daten
- Benennung der C-Deklarationen
- Erzeugung der Compilezeit-Repräsentation aus Java-Quelltext
- Erzeugung der C-Deklarationen

Instanz-Structs in Rt_tables

```
type inst_entry =
  Inst_field of Annot.field
type inst = {
  inst_decl_nm : string;
  mutable inst_vtab : vtab;
  inst_entries : inst_entry list;
}
```

- inst_decl_nm: Der Name des C-Structs
- inst_vtab: Die virtuelle Tabelle, auf die das _vtab-Feld zeigen soll
- inst_entries: Bisher enthalten Instanzen nur Instanzvariablen (Felder)
- Wieder: Compilezeitumgebung

Klassen im Layout

- Modul Layout berechnet virtuelle Tabellen und Instanz-Structs
- Kontext: do_syntab ruft do_cls_nm und do_cls auf

```
let do_syntab syntab =
  iter (function Cls cls -> do_cls_nm cls) syntab.syntab_elems;
  iter (function Cls cls -> do_cls cls) syntab.syntab_elems
```

- do_cls_nm setzt bestimmt Namen für Klasse auf C-Ebene
- do_cls setzt Properties für virtuelle Tabelle und Instanz-Struct

Namen für Klassen

```
let nm_of_qid ?(sep = "$") ?n =
  let rec loop = function
    Ast.Qid id -> nm_of_id ~sep ?n id
    | Ast.Qid_suffix(qid,id) ->
      loop qid ^ sep ^ nm_of_id id
  in loop

let do_cls_nm cls =
  set_cls_nm cls (nm_of_qid cls.cls_name)
```

- Qualifizierter Klassenname (mit Paket), alle . durch \$ ersetzt
- Kann im Rest des Layout-Moduls verwendet werden

Eindeutige Namen für Felder

```
let nm_of_field f =
  cls_nm f.field_cls ^ "_" ^ nm_of_id f.field_name
```

- Feldnamen müssen
 - innerhalb des Instanz-Structs für Instanzvariablen
 - global für Klassenvariablen

eindeutig sein

⇒ Benutze als Präfix den C-Namen der Klasse, in der `f` deklariert ist

Eindeutige Namen für Felder und Methoden

- Felder und Methoden müssen in C eindeutige Namen bekommen
 - Komponente in Instanz-Struct
 - Methodename in Virtueller Tabelle
 - Funktionsname für Implementierung
- Möglichkeit 1: Durchnummerieren pro Klasse Methoden
 - ✗ Ergebnis abhängig von Reihenfolge in Eingabe
 - ✗ Ergebnis abhängig von Details des Compilers → nicht portabel
- Möglichkeit 2 ([name mangling](#)):
Eindeutige Zusätze wählen, die sich aus der Eingabe selbst ergeben
 - Klassennamen als Präfix zu Feldern, Methoden, Konstruktoren
 - Kürzel für Argumente zu Methoden und Konstruktoren

Einschub: Überladung

```
class PrintStream {
  void println() { ... }
  void println(int i) { ... }
  void println(String s) { ... }
  void println(Object s) { ... }
```

- Mehrere Methoden einer Klasse dürfen denselben Namen haben, solange sie nur unterschiedliche Parameterlisten besitzen (JLS §8.4.7).
- Beim Methodenaufruf werden alle passenden und erreichbaren Methoden zusammengestellt und dann die am besten passende ausgewählt (JLS §15.12.2).
- In C können Funktionen nicht überladen werden

⇒ Wir müssen selbst eindeutige Namen wählen

Eindeutige Methodennamen im Java-Bytecode

Idee: Methoden haben eindeutige [Signatur](#), die ihre Parametertypen und Rückgabe kodiert.

```
class Sigs {
  int f()      { return 0; }
  void f(int i) { }
  void f(Sigs s) { }
}
> javap -s Sigs
Compiled from "Sigs.java"
class Sigs extends java.lang.Object{
  Sigs();
    Signature: ()V
  int f();
    Signature: ()I
  void f(int);
    Signature: (I)V
  void f(Sigs);
    Signature: (LSigs;)V
}
```

Kurznamen für Typen

```
let nm_of_rty = function
  Rty_cls cls -> cls_nm cls

let nm_of_ty = function
  ...
  | Ty_int      -> "I"
  | Ty_char    -> "C"
  | Ty_boolean -> "B"
  | Ty_ref rty -> "R"^(nm_of_rty rty)

let nm_of_formals fs =
  String.concat ""
  (map (fun f -> nm_of_ty f.formal_ty) fs)
```

Eindeutige Namen für Methoden

```
let nm_of_meth m =
  nm_of_id m.meth_name ^ "_" ^
  nm_of_formals m.meth_formals ^ "_" ^
  nm_of_ty m.meth_ret

let nm_of_meth_function m =
  (match m.meth_decl_in with
   | Rty_cls cls -> cls_nm cls) ^
  "_" ^
  nm_of_meth m
```

- Java-Methodennamen sind nicht eindeutig in einer Klasse → Überladung
- Verwende als Suffix Kurznamen für die Parameter- und Rückgabetypen
- Berechne auch global eindeutigen Name für C-Implementierungsfunktion

Eindeutige Namen für Konstruktoren

```
let nm_of_ctor c =
  cls_nm c.ctor_cls ^ "_ctor_" ^
  nm_of_formals c.ctor_formals
```

- Konstruktoren sind in Java-Klasse nicht eindeutig → Überladung
- Verwende als Suffix Kurznamen für Parametertypen

Namen als Properties

- Funktion `Layout.do_cls` `cls` legt alle Details für Klasse `cls` fest
- Fügt Namen als Properties zu Methoden und Feldern hinzu
- Erzeugt schließlich virtuelle Tabelle (→ folgende Folien)

```
let rec do_cls cls =
  ...
  iter
  (fun f -> set_field_nm f (nm_of_field f))
  cls.cls_fields;
  iter
  (fun m ->
    set_meth_nm m (nm_of_meth m);
    set_meth_fun m (nm_of_meth_function m))
  cls.cls_meths;
  ...
  set_cls_vtab cls (create_cls_vtab cls)
  set_cls_inst cls (create_cls_inst_struct cls);
```

Aufbau Virtuelle Tabelle

```
let create_cls_vtab cls =
  let entries_for cls = ...
  in
  {
    vtab_decl_nm = cls_nm cls ^ nm_sep ^ "vtab";
    vtab_def_nm = cls_nm cls ^ nm_sep ^ "vtab";
    vtab_entries = entries_for cls;
  }
```

- Rückgriff auf `cls_nm` Property für Benennung
- Benennung von Struct und globaler Variable
- Erzeugung von Einträgen delegiert an lokale Hilfsfunktion

Zwischenstand

- ✓ Laufzeit-Datenstruktur für Objekte → Virtuelle Tabellen
- ✓ Compilezeit-Repräsentation dieser Daten
- ✓ Benennung der C-Deklarationen
- Erzeugung der Compilezeit-Repräsentation aus Java-Quelltext
- Erzeugung der C-Deklarationen

Aufbau Virtuelle Tabelle

```
let entries_for cls =
  mapfilter
  (fun m ->
    if is_static_meth m
    then None
    else Some(Vtab_meth {
      vtm_decl = m;
      vtm_def = m;
    }))
  cls.cls_meths
```

- Detail: Statische Methoden erhalten keine Einträge
→ Vergleiche statische Felder auf Blatt 7
- Für neue Methoden sind Deklaration und Definition gleich

Beispiel: Klasse Point

```
struct Point$Vtbl {
    void (* moveBy_II_V)();
    int (* getX__I)();
    int (* getY__I)();
};
void Point_moveBy_II_V(struct Point$obj* __this,int dx,int dy);
int Point_getX__I(struct Point$obj* __this);
int Point_getY__I(struct Point$obj* __this);
```

- Virtuelle Tabelle enthält Zeiger auf Funktionen
- (Parametertypen sind ausgelassen, um später Warnungen zu verhindern.)
- Methodennamen in Virtueller Tabelle sicher eindeutig
- Globale Implementierungsfunktionen eindeutig benannt

Beispiel: Klasse Point

```
struct Point$obj {
    struct Point$Vtbl* _Vtbl;
    int Point_x;
    int Point_y;
};
```

- Instanz-Struct enthält
 - Instanzvariablen
 - Verweis auf die virtuelle Tabelle
- Namen der Felder sind eindeutig.
- (In Java können abgeleitete Klassen Felder verdecken, indem sie ein eigenes Feld mit dem gleichen Namen deklarieren (JLS, §8.3). Der Klassenname wird hier also wirklich benötigt.)

Berechnung der Instanz-Structs

```
let create_cls_inst_struct cls =
  let entries_for cls =
    mapfilter
      (function f when is_static_field f -> None
       | f -> Some(Inst_field f))
      cls.cls_fields
  in (* body of create_inst_struct *)
  {
    inst_decl_nm = cls_nm cls ^ nm_sep ^ "obj";
    inst_vtab = cls_vtab cls;
    inst_entries = entries_for cls;
  }
```

- Jedes nicht-statische Feld wird Komponente von Instanz-Struct
- Benennung des Struct-Typs auf C-Ebene
- Eintrag der virtuellen Tabelle für Initialisierung: Property cls_vtab

Zwischenstand

- ✓ Laufzeit-Datenstruktur für Objekte → Virtuelle Tabellen
- ✓ Compilezeit-Repräsentation dieser Daten
- ✓ Benennung der C-Deklarationen
- ✓ Erzeugung der Compilezeit-Repräsentation aus Java-Quelltext
- Erzeugung der C-Deklarationen

Codegenerator für Typen

```
let cg_tydecl decl =
  match decl with
  | Cls cls ->
    let (hdr,impl) = cg_cls cls in
    [ prog_of_code
      ~file: ((file_for_cls cls)~".h")
      ~guard: (string_of_qidentifier cls.cls_name)
      hdr;
      prog_of_code
      ~file: ((file_for_cls cls)~".c")
      impl
    ]
```

- Pro Klasse eine Header- und eine Implementierungsdatei
- Aufgabe pro Klasse
 - Direkte Umsetzung von Virtuellen Tabellen und Instanzen in C-Structs
 - Übersetzung von Methodenrümpfen

OCaml-Hinweis: ~file, ~guard sind benannte optionale Parameter

Code für Virtuelle Tabelle

```
let decl_vtab vtab =
  C.Decl_ty(C.Ty_struct(
    Some vtab.vtab_decl_nm,
    (map (function Vtab_meth m ->
      let dm = m.vtm_decl in
      (meth_nm dm,
        ty_fun_ptr dm.meth_ret))
      vtab.vtab_entries)))
let decl_cls_vtab cls = decl_vtab (cls_vtab cls)
```

Erzeugt beispielsweise:

```
struct Point$vtab {
  void (* moveBy_II_V)();
  int (* getX__I)();
  int (* getY__I)();
};
```

Hinweis: Blatt 7, Aufgabe 1 enthält diese Funktion im kleinen

Codegenerator für Klassen

```
let cg_cls cls =
  let vtab_decl = decl_cls_vtab cls
  and vtabs = def_cls_vtabs cls
  and inst_decl = decl_cls_inst cls
  and incls = map include_for_rty cls.cls_refs
  ...
  in let hdr = Code(incls, [ vtab_decl; inst_decl ] @ ...)
  and impl = Code(incls, ... @ vtabs ...)
  in (hdr,impl)
```

- Header enthält C-Structs für Virtuelle Tabelle und Instanz-Struct
- impl enthält globale Variable für virtuelle Tabelle (Hinweis: Der Plural vtabs wird erst für Interfaces benötigt.)
- Includes binden Header für benutzte Klassen ein

Code für Virtuelle Tabelle

```
let def_for_vtab vtab =
  C.Decl_var(some vtab.vtab_def_nm,           // name of var
    C.Ty_struct_ref vtab.vtab_decl_nm,      // type of var
    Some(C.Exp_array_init                   // initializer
      (map (function Vtab_meth m ->
        C.Exp_var (meth_fun (some m.vtm_def))
          vtab.vtab_entries)))
  )
let def_cls_vtabs cls =
  def_for_vtab (cls_inst cls).inst_vtab
```

Erzeugt beispielsweise:

```
struct Point$vtab Point$vtab =
{
  Point_moveBy_II_V,
  Point_getX__I,
  Point_getY__I
};
```

Zwischenstand

- Wir haben Klassen durch den gesamten Compiler gebracht!
 - Eindeutige Namen für Klassen, Felder, Methoden
 - Objektmodell mit virtuellen Tabellen
 - Compilezeit-Repräsentation von Laufzeitdaten
 - C-Repräsentation von Laufzeitdaten
- Noch zu tun bleiben
 - Instanziierung
 - Typcheck
 - Methoden-Rümpfe übersetzen