

Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

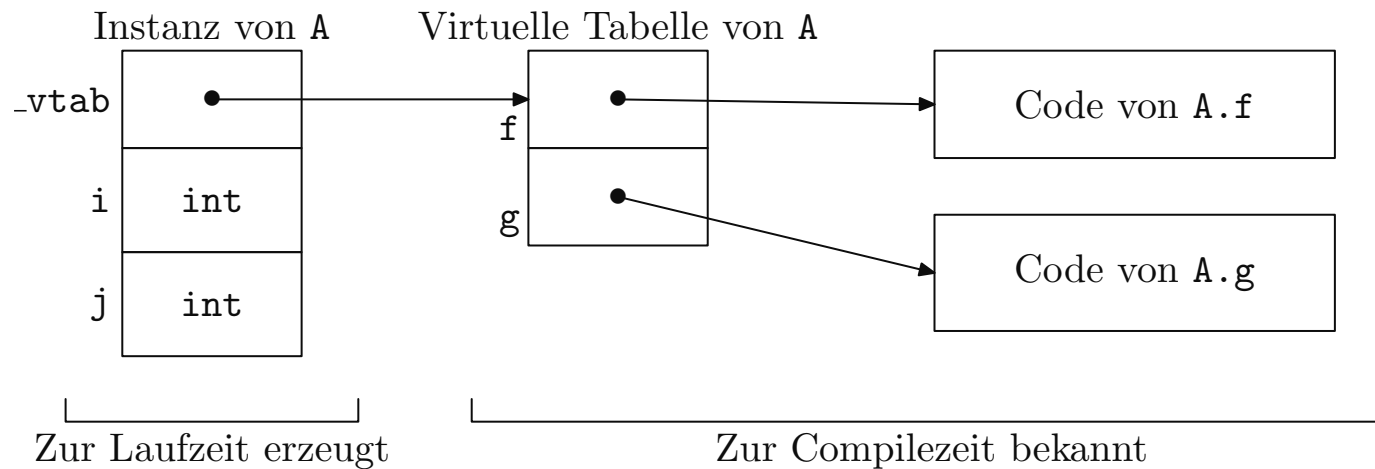
1. Dezember 2005

Zusammenfassung der letzten Stunde

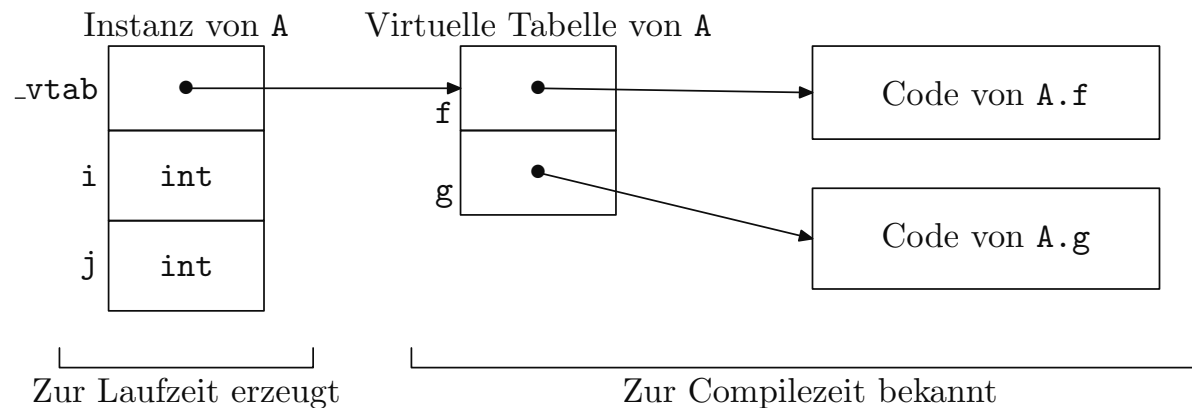
- Objektmodell mit virtuellen Tabellen
 - Eindeutige Namen für alle C-Konstrukte gewählt
 - Instanz-Structs und Virtuelle Tabellen in OCaml
 - Codeerzeugung für diese Structs
- Heute:
 - Nachtrag: Code für virtuelle Tabellen generieren
 - Methodenrümpfe übersetzen
 - Typcheck für durchführen

Erinnerung: Virtuelle Tabelle

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



Erinnerung: Ziel Methodenaufruf



`a.f()` \rightarrow `(t=a, t->_vtab->f(t))`

- Funktionszeiger aus virtueller Tabelle von Objekt a
- Aufruf mit `this`-Objekt wie bisher
- ✓ Jede Instanz kann anderen Code für dieselbe Methode ausführen lassen
- ✓ Polymorphie bei Überschreiben funktioniert

Erinnerung: Virtuelle Tabelle

Rt_tables

```
type vtab = {  
  mutable vtab_decl_nm : string;  
  mutable vtab_def_nm  : string;  
  mutable vtab_entries : vtab_entry list;  
}  
type vtab_entry = Vtab_meth of vtab_meth  
  
type vtab_meth = {  
  vtm_decl      : Annot.meth;  
  mutable vtm_def : Annot.meth;  
}
```

Erinnerung: Instanz-Structs

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;  
}
```

Nachtrag: Codegenerator für Typen

Cg

```
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

Nachtrag: Codegenerator für Klassen

Cg

```
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

Nachtrag: Code für Virtuelle Tabelle

Cg

```
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

Nachtrag: Code für Virtuelle Tabelle

Cg

```
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
 - Typcheck
 - Methoden-Rümpfe übersetzen

Erinnerung: Typen und Typsysteme

- Typen und Typsysteme verhindern Laufzeitfehler, die durch unerwartete Daten verursacht würden.
- Typen beschreiben Laufzeit-Datenstrukturen
 - Objekte oder primitive Typen
 - Vorhandene Felder und Methoden in Objekten
 - Daten in Feldern von Objekten
 - Parameter- und Rückgabe von Methoden
- Typsystem beschreibt das Laufzeitverhalten des Programms
 - Mit welchen Argumenten kann eine Methode fehlerfrei arbeiten?
 - Was passiert beim Schreiben in eine Variable?
 - Wie sieht eine mit `new` erzeugte Instanz aus?
 -

Erinnerung: Typregeln

- Eine Typregel beschreibt, wie sich ein bestimmtes Programmkonstrukt bei Ausführung in Hinblick auf die Typen verhalten wird.
- Beispiel: `new C(e1, ..., en)`
 - Laufzeit: **Wenn** C einen Konstruktor besitzt, der mit den Ergebnissen von e_1, \dots, e_n als Argumenten fehlerfrei abläuft, **dann** ist das Ergebnis eine Instanz der Klasse C .
 - Typregel: **Wenn** die Ergebnisse von e_1, \dots, e_n die Typen t_1, \dots, t_n haben **und** C einen Konstruktor mit formalen Parametern dieser Typen deklariert, **dann** ist hat das Ergebnis des Gesamtausdrucks den Typ C . (Die Klasse wird als Typ verwendet).

Typregeln und Übersetzung

- Typregeln beschreiben Laufzeitverhalten von Konstrukten
- ⇒ Um Typregeln zu verstehen, müssen wir die Bedeutung eines Konstrukts kennen
- ⇒ Um Typregeln genau zu verstehen, übersetzen wir am besten nach C (denn das Verhalten von C haben verstehen wir bereits)
- Ideal: Zuerst Übersetzung, dann Typregel entsprechend
- Aber: Codegenerator benutzt Ergebnisse des Typcheckers
- Der Plan:
 - Parallel Übersetzung und Typregeln für Anweisungen und Ausdrücke
 - ⇒ Betrachte gleichzeitig Module T_c und C_g

Konversionen (JLS §5)

- Situation
 - Wert v hat Typ s (bei Variablen z.B. per Deklaration)
 - v wird in einem Kontext verwendet, in dem ein Typ t erwartet wird
- Mögliche Lösungen
 - Fehlermeldung wegen falschem Typ → Anstrengend für Programmierer
 - Der Compiler versucht automatisch, v als einen neuen Wert v' darzustellen, der dann den gewünschten Typ t hat.
- Solche Umwandlungen heißen **Konversionen**

Arten der Konversion (JLS §5.1)

- Identity conversion

A conversion from a **type to that same type** is permitted for any type [, thus] allowing the simply stated rule that **every expression is subject to conversion**, if only a trivial identity conversion.

→ Vereinfacht die Implementierung

- Widening Primitive Conversion: Ersetzt Primitive Datentypen durch solche mit größerer Genauigkeit (\approx mehr Bits für die Darstellung)
- Narrowing Primitive Conversions: Ersetzt Primitive Datentypen durch solche mit kleinerer Genauigkeit (\approx weniger Bits für die Darstellung)
→ Informationsverlust

Arten der Konversion (JLS §5.1)

- Widening Reference Conversions: Ersetzt Typ durch eine Superklasse oder ein implementiertes Interface
→ werden wir später bei Vererbung behandeln
- Narrowing Reference Conversions → downcasts, bei Vererbung
- String Conversions: Umwandlung in einen String
→ Behandeln wir nicht

Tc: Konversionen

Tc

```
let conv a b =  
  match (a,b) with  
  | (a,b) when a == b -> Conv_none      (* identity *)  
  | Ty_int,Ty_float   -> Conv_prim(a,b)  
  | Ty_null, Ty_ref b -> Conv_none  
  | a, Ty_void        -> Conv_discard a  
  | _ -> type_error ("cannot convert " ^ string_of_ty a ^  
                    " to " ^ string_of_ty b)
```

- Funktion `conv` berechnet falls möglich eine Konversion (`Typ Conv.conv`)
- Nimmt an, dass die Konversion gebraucht wird und wirft Fehler
- Nur auszugsweise dargestellt
- `Ty_null` ist der Typ des `null` Literals – es hat jeden Typ

Prop: Konversionen

- Property `exp_conv` `e` hält fest, welche Konversion auf das Ergebnis von `e` angewandt werden muss, damit es im Kontext von `e` richtig benutzt werden kann
- Wird im Typchecker berechnet
- Ist eventuell `Conv_none` (die identity conversion)

Cg: Konversionen

```
let patch_conv_to_exp conv exp =  
  match conv with  
  | Conv_none           -> exp  
  | Conv_discard _     -> C.Exp_cast(C.ty_void, exp)  
  | Conv_prim(src,dst) -> C.Exp_cast(repr_ty dst,exp)
```

- Konversion erfordert Umwandlung von Wert zur Laufzeit
- Cg Modul muss entsprechenden C-Code erzeugen
- Hilfsfunktion `patch_conv_to_exp` fügt Cast-Ausdruck $(t)e$ ein

⇒ Verwendet vorhandene C-Konversionen

Cg: Methoden von Klassen

Cg

```
let cg_cls cls =  
  let meths    = map cg_meth cls.cls_meths  
  ...  
  in let hdr   = Code(incls, ... @ (map C.decl_of_def meths) @ ..  
    and impl  = Code(incls, ... @ meths @ ...)  
  in (hdr,impl)
```

- Die C-Funktionen für Methoden enthalten ausführbaren Code
- Deklaration in Header-Datei, Definition in C-Datei
(Aufteilung beim letzten Mal besprochen)

Tc: Klassen

Tc

```
let chk_tydecls decls = iter chk_tydecl decls
let chk_tydecl = function Cls cls -> chk_cls cls
let chk_cls cls =
  iter chk_meth  cls.cls_meths;
  iter chk_ctor  cls.cls_ctors;
  iter chk_init  cls.cls_inits;
  iter chk_field cls.cls_fields
```

- Alle Bestandteile von Klassen unterliegen Typregeln
 - Typregeln beziehen sich nur auf ausführbaren Code
- ⇒ `chk_meth` enthält die Hauptarbeit, die anderen sind leichte Variantionen
- ⇒ Wir behandeln hier nur `chk_meth`

Methoden übersetzen

- Jede Methode wird zu einer C-Funktion
- Der Name ist `meth_nm Property` (→ letzte Stunde)
- Rumpf der Methode in `meth_body` ist eine Anweisung (`Annot.sm`)
- Instanzmethoden bekommen einen `this`-Parameter

```
void Point_moveBy_II_V(struct Point$obj* __this,
                      int dx$1,int dy$2) {
    {
        (void )(((__this)->Point_x)=(((__this)->Point_x)+(dx$1)));
        (void )(((__this)->Point_y)=(((__this)->Point_y)+(dy$2)));
    }
}
```

Einschub: Namen für lokale Variablen und Parameter

Cg

```
let local_nm_suffix = ref 0
let next_local_nm id =
  incr local_nm_suffix;
  nm_of_id ~n: !local_nm_suffix id

let assign_formals_nm formals =
  iter
    (fun f -> set_formal_nm f (next_local_nm f.formal_id))
    formals

let assign_locals_nm locals =
  iter
    (fun local ->
      set_local_nm local (next_local_nm local.local_id))
    locals
```

- Layout hat keine C-Namen für lokale Variablen und Parameter festgelegt
- next_local_nm hängt an ein Annot.id eine fortlaufende Nummer an

Einschub: Temporäre Variablen

- Modul Temps in Modul Cg verwaltet temporäre Variablen
- `val reset : unit -> unit` fängt mit der Vergabe neu an
- `val used : unit -> C.decl list` ergibt benutzte Variablen
- `val alloc : unit -> C.exp` wählt nächste freie Variable
- Ziel: Aktuelle Argumente nach Funktionsaufruf wieder vergessen.

⇒ Anforderung und Freigabe in Gruppen

- `val begin_group : unit -> unit` beginnt eine neue Gruppe
- `val end_group : unit -> unit` gibt aktuelle Gruppe frei
- `val reset_groups : unit -> unit` beendet alle Gruppen

Cg: Parameterliste

Cg

```
let repr_formals cls static formals =  
  let decls =  
    map  
      (fun f -> (formal_nm f, repr_ty f.formal_ty))  
      formals  
  in  
    if static  
    then decls  
    else ("__this", ty_ptr_to_cls_inst_struct cls) :: decls
```

- Hauptarbeit: Repräsentation der formalen Parameter
- Für Instanzmethoden einen this-Parameter hinzufügen
- Verwendet in `cg_meth` und `cg_ctor`

Cg: Methoden

Cg

```
let cg_meth m = (* wird nur für Methoden in Klassen aufgerufen *)
  let cls = rty_as_cls m.meth_decl_in in
  match m.meth_body with
  | None -> None (* kein Rumpf *)
  | Some sm ->
    assign_formals_nm m.meth_formals;
    Temps.reset();
    let csm = cg_sm sm in
    let locals = unique csm.csm_locals
    and formals = repr_formals cls (is_static_meth m)
                                     m.meth_formals
    and ret = repr_ty m.meth_ret
    and vars = Temps.used() @ locals
    in
      Some(C.Decl_fun(meth_fun m,
                     formals, ret,
                     (Some(C.Sm_block(vars, [ csm.csm_sm ]))))))
```

Cg: Methoden

```
void Point_moveBy_II_V(struct Point$obj* __this,  
                       int dx$1,int dy$2) {  
    {  
        (void )(((__this)->Point_x)=(((__this)->Point_x)+(dx$1)));  
        (void )(((__this)->Point_y)=(((__this)->Point_y)+(dy$2)));  
    }  
}
```

- Direkt sichtbar
 - Name mangling
 - this-Objekt
 - Formale Parameter mit eindeutigen Namen

Tc: Methoden

Tc

```
let chk_meth m =  
  match m.meth_body with  
  | None      -> () (* nothing to be done *)  
  | Some sm   -> chk_sm sm
```

- Der gesamte ausführbare Code liegt im Rumpf

⇒ Typüberprüfung nur für den Rumpf der Methode

Anweisungen

- Anweisungen sind `Annot.sm` Knoten
- Übersetzung in C-Statements `C.sm`
- Vgl. auch “erster Versuch” vom 22.11.
- Anweisungen behandeln selbst keine Daten → Typcheck trivial
- Ausnahme: `if` und `while` benötigen `boolean` Ausdrücke

Code für Anweisungen

Cg

```
type csm = {  
  csm_sm      : C.sm;  
  csm_locals  : C.decl list;  
}
```

- C-Übersetzung der Anweisung in `csm_sm`
- Notwendige Deklarationen für lokale Variablen
→ Werden am Anfang des umgebenden Blocks eingefügt

Cg: Return-Anweisung

Cg

```
let cg_sm sm =
  match sm.sm_desc with
  | Sm_return_void ty ->
    { csm_sm = C.Sm_return;
      csm_locals = [];
    }
  | Sm_return_exp(ty,e) ->
    { csm_sm = C.Sm_return_exp(cg_exp e);
      csm_locals = [];
    }
  | _ -> not_implemented_sm sm None
```

- Direkte Übersetzung
- Fall `return_void` liefert keinen Rückgabewert in C
- Fall `return_exp` liefert das Ergebnis eines Ausdrucks

Beispiel: Return-Anweisung

```
public int getX() {  
    return x;  
}
```

wird zu

```
int Point_getX__I(struct Point$obj* __this) {  
    {  
        return (__this)->Point_x;  
    }  
}
```

Tc: Return-Anweisung

```
let chk_sm sm =  
  match sm.sm_desc with  
  | Sm_return_void ty ->  
    if ty != Ty_void  
    then type_error "return without value in non-void method"  
  | Sm_return_exp(ty,e) ->  
    set_exp_conv e (conv (chk_exp e) ty)
```

- Fehlermöglichkeit: nicht-void Methode liefert keinen Rückgabewert
- Falls Methode einen Rückgabewert verspricht, dann muß das e einen Wert mit entsprechendem Typ liefern
- ⇒ conv wirft Typfehler wenn keine Konversion vorhanden
- ⇒ Benötigte Konversion wird als Property exp_conv festgehalten
- ✓ Typcheck stellt sicher: Zur Laufzeit wird erlaubter Wert zurückgegeben

Cg: Ausdruck als Anweisung

Cg

```
let cg_sm sm =  
  match sm.sm_desc with  
  | Sm_exp e ->  
    Temps.reset_groups();  
    in { csm_sm = C.Sm_exp(cg_exp e);  
        csm_locals = [];  
        }
```

- Initialisiere Temps: Momentan keine temporären Variablen belegt
- Übersetze e in einen C-Ausdruck C.exp
- Fasse in C.Sm_exp ein → Verwirft Ergebnis

Tc: Ausdruck als Anweisung

Tc

```
let chk_sm sm =  
  match sm.sm_desc with  
    Sm_exp e -> set_exp_conv e (conv (chk_exp e) Ty_void)
```

- Typprüfung für e
- Verwerfe das Ergebnis durch Einfügen einer Conv_discard
- ✓ Typcheck berechnet Laufzeitverhalten voraus

Cg: Lokale Variablen

Cg

```
let cg_sm_block = function
  | Smb_local locals ->
    assign_locals_nm locals; (* Namen für C-Prog generieren *)
    { csm_locals = (* neue Deklarationen für den Block *)
      map (fun local ->
          C.Decl_var(local_nm local,
                    repr_ty local.local_ty,
                    Some(C.Exp_const_int 0)))
        locals;
      csm_sm = (* neue Anweisungen für den Block *)
        (C.Sm_seq (mapfilter
          (fun local ->
            match local.local_init with
            | None -> None
            | Some e ->
              Some(C.Sm_exp(C.Exp_assign(
                C.Exp_var (local_nm local),
                cg_exp e))))
          locals)) ) }
```

Beispiel: Initialisierte lokale Variablen

```
void f() {  
    int i = 5;  
    i = i + 1;  
    int k = i;  
    { int n = k;  
    }  
}
```

```
void LocalInit_f__V(struct LocalInit$obj* __this) {  
    int k$2 = 0;;  
    int i$1 = 0;;  
    (i$1)=(5);  
    (void )((i$1)=((i$1)+(1)));  
    (k$2)=(i$1);  
    { int n$3 = 0;;  
        (n$3)=(k$2);  
    }  
}
```

- Lokale Variable im aktuelle Block am Anfang deklariert
- Initialisierung ist Zuweisung an der Stelle, an der Java-Deklaration stand.

Tc: Blockanweisung

```
let chk_sm sm =  
  match sm.sm_desc with  
  | Sm_block smbs -> iter chk_sm_block smbs  
and chk_sm_block = function  
  Smb_local locals -> iter chk_local locals  
  | Smb_sm sm -> chk_sm sm  
and chk_local l =  
  match l.local_init with  
  | None -> ()  
  | Some e -> set_exp_conv e (conv (chk_exp e) l.local_ty)
```

- Für Anweisung `sm`: Nichts zu tun, alle Berechnungen finden in `sm` statt, daher müssen auch nur dort Typchecks durchgeführt werden.
- Typregel für `local`: Der Wert, den die Initialisierung liefert, muß konvertierbar sein in den Typ der lokalen Variablen
- ✓ Typregel stellt sicher, daß Zuweisung zur Laufzeit erfolgreich sein wird

Ausdrücke übersetzen

- Jeder Java-Ausdruck wird zu einem C-Ausdruck
 - Das Ergebnis des Ausdrucks wird das Ergebnis des C-Ausdrucks
 - Hintereinanderausführung durch C-Sequenzausdrücke
 - Property `exp_ty` enthält (berechneten) Typ des Ergebnisses
 - `Tc` berechnet auch benötigte Konversion in Property `exp_conv`
- ⇒ Ab jetzt müssen wir `Tc` vor `Cg` behandeln

Tc: Ausdruck

Tc

```
let keep_exp_ty e ty =  
  set_exp_ty e ty;  
  ty
```

```
let rec chk_exp e =  
  keep_exp_ty e  
  (match e.exp_desc with  
   ...)
```

- Fallunterscheidung nach der Art des Ausdrucks

⇒ Im folgenden die Fälle von `Exp_...` durchgehen

- Jeder Fall liefert den Typ des Ergebnisses von `e`

⇒ Speichern in Property `exp_ty`

Cg: Ausdruck

Cg

```
let rec cg_exp exp =  
  patch_conv_to_exp (exp_conv exp)  
  (match exp.exp_desc with  
    ...)
```

- Fallunterscheidung nach Art des Ausdrucks
 - Tc hatte Property “muss noch konvertiert werden” hinterlassen
- ⇒ Code einfügen, um benötigte Konversion durchzuführen

Tc: Bezeichner

Tc

```
Exp_id eid ->
  (match eid with
   | Eid_field f   -> f.field_ty
   | Eid_cls      c   -> Ty_static c
   | Eid_local l   -> l.local_ty
   | Eid_formal f -> f.formal_ty)
```

- Das Ergebnis des Ausdrucks ist der in der Variable gespeicherte Wert

⇒ Deklarierter Typ ist Typ des Ausdrucks

- Analog: Literale

Cg: Bezeichner

Cg

```
Exp_id exp_id ->
  match exp_id with
  | Eid_formal f -> C.Exp_var (formal_nm f)
  | Eid_local l -> C.Exp_var (local_nm l)
  | Eid_field f ->
    if is_static_field f
    then C.Exp_var (field_nm f)
    else C.Exp_deref_acc(repr_this (), field_nm f)
```

- Zugriff auf entsprechende C-Variable
- ✓ Ergebnis ist wie vom Typchecker angenommen
- Feldzugriff für Instanz- und Klassenvariablen getrennt
- Analog: Literale

Tc: Argumente \rightarrow Parameter

Tc

```
let chk_args_against_formals args formals =  
  if length args != length formals  
  then type_error "number of formals != actuals";  
  iter2  
    (fun act form ->  
      set_exp_conv act (conv (exp_ty act) form.formal_ty))  
  args  
  formals
```

- Bei Methoden- und Konstruktoraufruf
 - Anzahl der Argumente und Parameter prüfen
 - Für jedes Argument entsprechende Konversion einfügen
- Annahme: Typen von Argumenten schon zuvor berechnet

Tc: Instanziierung

Tc

```
Exp_new(Rty_cls cls, args) ->
  let _ = map chk_exp args in
  let ctor = find_called_ctor cls args
  in
    set_exp_acc_ctor e ctor;
    chk_args_against_formals args ctor.ctor_formals;
    Ty_ref (Rty_cls cls)
```

- Zuerst Typen der Argumente berechnen
- Dann besten passenden Konstruktor finden (und als Property speichern)
(→ Überladung, noch nicht implementiert)
- Schließlich Konstruktoraufruf prüfen
- Ergebnis: Ein Objekt der instanziierten Klasse

Cg: Instanziierung

```
Exp_new(Rty_cls cls, args) ->
  let _ = Temps.begin_group() in
  let args' = map cg_exp args in
  let _ = Temps.end_group() in
  let ctor = exp_acc_ctor exp in
  let nobj = Temps.alloc() in
  C.Exp_seq [
    C.Exp_assign(nobj, C.Exp_call(C.Exp_var (cls_alloc cls), [])),
    C.Exp_call(C.Exp_var (ctor_fun ctor), nobj :: args');
  ]
  nobj
]
```

- Argumente in neue temporäre Variable auswerten
- alloc-Funktion der Klasse aufrufen
- Ausgewählten Konstruktor mit neuem Objekt aufrufen
- Bemerkung: Konstruktor wird nicht über virtuelle Tabelle ausgewählt
- ✓ Ergebnis ist neues Objekt der Klassen cls wie von Tc berechnet

Beispiel: Instanziierung

```
Point p = new Point(10,20);
```

```
(p$6)=(((__tmp_0)=(Point$alloc()),  
        Point_ctor_II(__tmp_0,10,20),  
        __tmp_0));
```

Tc: Methodenaufruf

```
Exp_call(e',id,args) ->
  let ty_e' = chk_exp e' in
  let _ = map chk_exp args in
  let (formals,ret) =
    (match ty_e' with
     | (Ty_ref (Rty_cls cls)) ->
       let m = find_called_cls_meth cls id args in
         set_exp_acc_meth e m;
         (m.meth_formals, m.meth_ret)
    )
  in
  chk_args_against_formals args formals;
  ret
```

- Verwende den Typ von e' um die verfügbaren Methoden zu finden
- Wähle die passendste Methode aus (und setze Property exp_acc_meth) (→ Überladung, noch nicht implementiert)
- Prüfe Argumente für die ausgewählte Methode
- Ergebnistyp ist deklarierter Rückgabotyp

Cg: Methodenaufruf

```
| Exp_call(e',id,args) ->
  let _ = Temps.begin_group() in
  let exp' = cg_exp e' in
  let this = Temps.end_and_alloc () in
  let _ = Temps.begin_group() in
  let args' = map cg_exp args in
  let argtmp' = map (fun _ -> Temps.alloc()) args in
  let argass = map2 (fun t a->C.Exp_assign(t, a)) argtmp' args' :
  let _ = Temps.end_group() in
  let meth = exp_acc_meth exp in
  ...
```

- Werte e' und args aus und reserviere dabei temporäre Variablen für die Ergebnisse dieser Ausdrücke
- Hole die vom Typchecker bestimmte aufzurufende Methode exp_acc_meth

⇒ Alle Daten für Codegenerierung vorhanden

Cg: Statischer Methodenaufruf

```
(match exp_ty e' with
  Ty_ref(Rty_cls cls) -> (* später: Interfaces *)
    if is_static_meth meth
    then C.Exp_seq
      (argass @ [C.Exp_call(C.Exp_var (meth_fun meth), argtmp')])
    else ...)
```

- Speichere Argumente in temporären Variablen → Auswertungsreihenfolge
- Wenn die ausgewählte Methode statisch ist
 - Es ist klar, welcher Code ausgeführt wird (!)
 - Der Aufruf hat kein `this` Argument
- Genau ein Funktionsaufruf in C:
 - Der Methodenname steht fest
 - Genau die Argumente werden übergeben
- ✓ Ergebnis: Rückgabe der Funktion, wie von Tc berechnet

Cg: Dynamischer Methodenaufruf

```
let inst = cls_inst cls in
  C.Exp_seq(
    [ C.Exp_assign(this, exp') ] @ argass @
    [ C.Exp_call(
      C.Exp_deref_acc(C.Exp_deref_acc(
        C.Exp_cast(C.Ty_ptr(C.Ty_struct_ref(inst.inst_decl_nm))
          this),
        "_vtab"), meth_nm meth),
      this :: argtmp') ])
```

- Speichere in `this` das Ergebnis von `exp'`
- Typcheck garantiert: `this` ist Zeiger auf Instanz von `cls`
- ⇒ C-Cast in `C.Exp_cast(...)` auf Instanz-Struct dieser Klasse erlaubt
- ⇒ Zugriff auf virtuelle Tabelle der Instanz erlaubt
- ⇒ Hole Funktionszeiger aus virtueller Tabelle der Instanz
- ✓ Ergebnis: Rückgabe der Funktion, wie von Tc berechnet

Beispielaufruf: getX()

```
System.out.println_int(p.getX());
```

```
((__tmp_0)=(java$lang$System_out),  
 (__tmp_2)=(((__tmp_1)=(p$6),  
             (((struct Point$obj*)(__tmp_1))->_vtab)  
             ->getX__I)(__tmp_1))),  
 (((struct java$lang$PrintStream$obj*)(__tmp_0))->_vtab)  
  ->println_int_I_V)(__tmp_0,__tmp_2));
```

Zusammenfassung

- Die wesentlichen OO-Konzepte sind geschafft!
 - Komplette Klassen übersetzt
 - Objekte mit virtuellen Tabellen
 - Instanziierung
 - Methodenaufruf
 - Genauer Verständnis des Typcheckers
 - Direkter Vergleich Module Tc und Cg
 - Der Typchecker berechnet voraus, was zur Laufzeit passiert
 - Der Codegenerator braucht Informationen vom Typchecker
 - Automatische Konversionen
 - Ausgewählte Methoden
- ⇒ Der Compiler erklärt im Detail, wie die Sprache Java funktioniert