Compilerbau Vorlesung WS 2008–11,13,15,17,19,SS 22,24

Johannes Waldmann, HTWK Leipzig

12. Mai 2024

Organisatorisches (Ü KW 15)

- Skript (Woche für Woche) `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
- Quelltexte aus VL, Diskussion Hausaufgaben
  `https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/cb-ss24` *Einschreiben!* wird dann *private*.
- autotool `https://autotool.imn.htwk-leipzig.de/new/vorlesung/325/aufgaben/aktuell` . *Einschreiben!*
- Wdhlg: Arbeiten im Pool: `$PATH`, `ghci` (9.8.2),
  mit git(lab.dit) (`ssh-keygen`, `.ssh/id_rsa.pub`)
- Haskell-Tooling (cabal, `ghc -haddock`, `:doc`)
- Diskussion Klausur PPS WS23 bei Bedarf

Beispiel: C-Compiler

- ```
  int gcd (int x, int y) {
    while (y>0) { int z = x%y; x = y; y = z; }
    return x; }
  ```
- `gcc -S -O2 gcd.c` erzeugt `gcd.s`:

```
.L3:    movl    %edx, %r8d ; cltd  ; idivl   %r8d
        movl    %r8d, %eax  ;   testl   %edx, %edx
        jg      .L3
```
```

Ü: was bedeutet `cltd`, warum ist es notwendig?

Ü: welche Variable ist in welchem Register?

- identischer (!) Assembler-Code für

```
int gcd_func (int x, int y) {
  return y > 0 ? gcd_func (y,x % y) : x;
}
```

- vollständige Quelltexte: siehe Repo
- Bsp Java-Kompilation: `https://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/`

Inhalt der Vorlesung

Konzepte von Programmiersprachen

- Semantik von einfachen (arithmetischen) Ausdrücken
- lokale Namen, • Unterprogramme (Lambda-Kalkül)
- Zustandsänderungen (imperative Prog.)
- Continuations zur Ablaufsteuerung

realisieren durch

- Interpretation, • Kompilation

Hilfsmittel:

- Theorie: Inferenzsysteme (f. Auswertung, Typisierung)
- Praxis: Haskell, Monaden (f. Auswertung, Parser)

Einleitung: Sprachverarbeitung

- mit Interpreter:
  - Quellprogramm, Eingaben $\overset{\text{Interpreter}}{\longrightarrow}$ Ausgaben
- mit Compiler:
  - Quellprogramm $\overset{\text{Compiler}}{\longrightarrow}$ Zielprogramm
  - Eingaben $\overset{\text{Zielprogramm}}{\longrightarrow}$ Ausgaben

- Mischform:
  - Quellprogramm $\overset{\text{Compiler}}{\longrightarrow}$ Zwischenprogramm
  - Zwischenprogramm, Eingaben $\overset{\text{virtuelle Maschine}}{\longrightarrow}$ Ausgaben
- reale Maschine (CPU) ist Interpreter für Maschinen- sprache (Interpretation in Hardware, in Microcode)
- gemeinsam ist: syntaxgesteuerte Semantik (Ausführung oder Übersetzung)

Literatur

- Franklyn Turbak, David Gifford, Mark Sheldon: *Design Concepts in Programming Languages*, MIT Press, 2008. `http://cs.wellesley.edu/~fturbak/`
- Guy Steele, Gerald Sussman: *Lambda: The Ultimate Imperative*, MIT AI Lab Memo AIM-353, 1976
  (the original 'lambda papers', `https://web.archive.org/web/20030603185429/http://library.readscheme.org/page1.html`)
- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd edition)* Addison-Wesley, 2007, `http://dragonbook.stanford.edu/`
- J. Waldmann: *Das M-Wort in der Compilerbauvorlesung*, Workshop der GI-Fachgruppe Prog. Spr. und Rechnerkonzepte, 2013 `http://www.imn.htwk-leipzig.de/~waldmann/talk/13/fg214/`

Anwendungen von Techniken des Compilerbaus

- Implementierung höherer Programmiersprachen
- architekturspezifische Optimierungen (Parallelisierung, Speicherhierarchien)
- Entwurf neuer Architekturen (RISC, spezielle Hardware)
- Programm-Übersetzungen (Binär-Übersetzer, Hardwaresynthese, Datenbankanfragesprachen)
- Software-Werkzeuge (z.B. Refaktorisierer)
- domainspezifische Sprachen

Organisation der Vorlesung

- pro Woche eine Vorlesung, eine Übung.
- in Vorlesung, Übung und Hausaufgaben:
  - Theorie,
  - Praxis: Quelltexte (weiter-)schreiben
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)

  Bei Interesse und nach voriger Absprache: Ersatz eines Teiles der Klausur durch vorherige Hausarbeit

  z.B. Reparaturen an autotool-Aufgaben oder anderem open-source-Projekt (Ihrer Wahl), bei denen Techniken des Compilerbaus angewendet werden

Beispiel: Interpreter f. arith. Ausdrücke

```
data Exp = Const Integer
         | Plus Exp Exp | Times Exp Exp
    deriving ( Show )

ex1 :: Exp
ex1 =
  Times ( Plus ( Const 1 ) ( Const 2 ) ) ( Const 3 )

value :: Exp -> Integer
value x = case x of
    Const i -> i
    Plus x y -> value x + value y
    Times x y -> value x * value y
```

das ist syntax-gesteuerte Semantik:
Wert des Terms wird aus Werten der Teilterme kombiniert

Beispiel: lokale Variablen und Umgebungen

```
data Exp = ... | Let String Exp Exp | Ref String
ex2 :: Exp
```

```
ex2 = Let "x" ( Const 3 )
     ( Times ( Ref "x" ) (Ref "x" ) )
type Env = ( String -> Integer )
extend n w e = \ m -> if m == n then w else e m
value :: Env -> Exp -> Integer
value env x = case x of
    Ref n -> env n
    Let n x b -> value (extend n (value env x) env) b
    Const i -> i
    Plus x y -> value env x + value env y
    Times x y -> value env x * value env y
test2 = value (\ _ -> 42) ex2
```

Bezeichner sind Strings — oder nicht?

- `... | Let String Exp Exp` — wirklich?
- es gilt `type String = [Char]`, also
  - einfach verkettete Liste von Zeichen
  - mit Bedarfsauswertung (lazy Konstruktoren)
- das ist
  - ineffizient (in Platz *und* Zeit)
  - egal (für unseren einfachen Anwendungsfall)
  - gefährlich (wenn man es für andere Anwendungen übernimmt)
- deswegen jetzt schon Diskussion . . .
  - von alternativen Implementierungen
  - und wie man diese versteckt

Datentypen für Folgen (von Zeichen)

- `type String = [Char]`: einfach verkettet, lazy: ist in den allermeisten Fällen unzweckmäßig
- `data Vector a`: Array (d.h., zusammenhängender Speicherbereich, deswegen effiziente Indizierung) mit kostenlosem *slicing* (Abschnitt-Bildung)
- `data Bytestring`: ≈ Vektor von Bytes (d.h., für rein binären Datenaustausch)

- `data Text` (aus Modul `Data.Text`) *efficient packed, immutable Unicode text type*, (d.h., Zeichen = Bytefolge)
- Modul `Data.Text.Lazy`: lazy Liste von (strikten) `Text`-Abschnitten, für Stream-Verarbeitung

Verstecken von Implementierungsdetails

- Implementierung direkt sichtbar:
  `data Exp = ... | Let Text Exp Exp`
- Verschieben der Implementierungs-Entscheidung:
  `type Id = Text; data Exp = ... | Let Id Exp Exp`
  bleibt aber sichtbar (`type`-Deklarationen werden bei Kompilation immer expandiert)
- Verstecken der Entscheidung: `modul Id (Id) where data Id = Id Text`
  exportiert wird Typ-Name, aber nicht der Konstruktor
  der Anwender (Importeur) von `Id` sieht `Text` nicht
- `data`-Deklaration mit genaue einem Konstruktor: erstetzen durch `newtype Id = Id Text`
  dieser kostet *gar nichts* (keine Zeit, keinen Platz)

Verwendung standardisierter Namen

- alle benötigten Funktionen (einschl. Konstruktoren) für `Id` implementieren und exportieren (es sind nicht viele)

  ```
  eqId::Id->Id->Bool; eqId (Id s) (Id t) = s == t
  ```

- diese spezifischen Namen will sich keiner merken ⇒ verwende standardisierte Typklassen, Bsp.

  ```
  instance Eq Id where (Id s) == (Id t) = s == t
  ```

  der Importeur von `Id` sieht den Namen `(==)` bereits, weil er in Prelude definiert ist

- wenn die Implementierung einer standardisierten Klasse eine einfache Delegation ist, kann sie vom Compiler erzeugt werden

  ```
  newtype Id = Id Text deriving Eq
  ```

Einsparung von Konstruktor-Aufrufen

- ```
      -- Implementierung des Konstruktors
  import qualified Data.Text as T
  fromString::String->Id;fromString s = Id (T.pack s)
      -- Anwendung:
  foo :: Id ; foo = fromString "bar"
  ```
- der Schreibaufwand wird verringert durch

  ```
           -- bei Implementierung:
  import Data.String;
  instance IsString Id where fromString = T.pack
           -- bei Anwendung:
  {-# language OverloadedStrings #-}
  foo :: Id ; foo = "bar"
  ```

  String-Literale sind dann *überladen* $\Rightarrow$ Compiler setzt `fromString` vor jedes (`"bar"`$\Rightarrow$`fromString "bar"`)

Übung (Haskell)

- Wiederholung Haskell
  - Interpreter/Compiler: ghci `http://haskell.org/`
  - Funktionsaufruf nicht `f(a,b,c+d)`, sondern `f a b (c+d)`
  - Konstruktor beginnt mit Großbuchstabe und ist auch eine Funktion
- Wiederholung funktionale Programmierung/Entwurfsmuster
  - rekursiver algebraischer Datentyp (ein Typ, mehrere Konstruktoren)
    (OO: Kompositum, ein Interface, mehrere Klassen)
  - rekursive Funktion
- Wiederholung Pattern Matching:
  - beginnt mit `case ... of`, dann Zweige
  - jeder Zweig besteht aus Muster und Folge-Ausdruck
  - falls das Muster paßt, werden die Mustervariablen gebunden und der Folge-Ausdruck auswertet
  ```

Übung (Interpreter)

- Benutzung:
  - Beispiel für die Verdeckung von Namen bei geschachtelten Let
  - Beispiel dafür, daß der definierte Name während seiner Definition nicht sichtbar ist
- Erweiterung:

  Verzweigungen mit C-ähnlicher Semantik:

  Bedingung ist arithmetischer Ausdruck, verwende 0 als Falsch und alles andere als Wahr.

```
data Exp = ...
         | If Exp Exp Exp
```

Übung (effiziente Imp. von Bezeichnern)

- welche Operationen auf `Id` werden benötigt?
  - Konstruktion (`fromString`)
  - Gleichheit
  - Ausgabe (nur für Fehlermeldungen!)
- für `newtype Id = Id Text deriving Eq`:
  wie teuer ist Vergleich? wie könnte man das verbessern?
- für `type Env` und `extend` wie angegeben: wie teuer ist das Aufsuchen des Wertes eines Namens in einer Umgebung, die durch n geschachtelte `extend` entsteht?
  wie könnte man das verbessern?
  Hinweis: mit `Env` als Funktion: gar nicht.
  Welcher andere Typ könnte verwendet werden?

1 Inferenz-Systeme

Motivation

- inferieren $=$ ableiten
- Inferenzsystem I, Objekt O,
  Eigenschaft $I \vdash O$ (in I gibt es eine Ableitung für O)
- damit ist I eine *Spezifikation* einer Menge von Objekten
- man ignoriert die *Implementierung* ($=$ das Finden von Ableitungen)
- Anwendungen im Compilerbau:
  Auswertung von Programmen, Typisierung von Programmen

Definition

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)
  Schreibweise $\frac{P_1,\ldots,P_n}{K}$
- Axiomen ($=$ Regeln ohne Prämissen)

eine *Ableitung* für F bzgl. I ist ein Baum:

- jeder Knoten ist mit einem Objekt beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von I
- Wurzel (Ziel) ist mit F beschriftet

Def: $I \vdash F : \Longleftrightarrow \exists\, I$-Ableitungsbaum mit Wurzel F.

Regel-Schemata

- um unendliche Menge zu beschreiben, benötigt man unendliche Regelmengen
- diese möchte man endlich notieren
- ein *Regel-Schema* beschreibt eine (mglw. unendliche) Menge von Regeln, Bsp: $\dfrac{(x, y)}{(x - y, y)}$
- Schema wird *instantiiert* durch Belegung der Schema-Variablen
  Bsp: Belegung $x \mapsto 13, y \mapsto 5$
  ergibt Regel $\dfrac{(13, 5)}{(8, 5)}$

Inferenz-Systeme (Beispiel)

- Grundbereich = Zahlenpaare $\mathbb{Z} \times \mathbb{Z}$
- Axiom: $\dfrac{}{(13, 5)}$
- Regel-Schemata: $\dfrac{(x, y)}{(x - y, y)}, \quad \dfrac{(x, y)}{(x, y - x)}$
- gilt $I \vdash (1, 1)$?
- Ü: Beziehung zu einem alten Algorithmus (früh im Studium, früh in der Geschichte der Menschheit)

Primalitäts-Zertifikate

- Satz: $p \in \iff \exists g : g$ ist *primitive Wurzel* mod p:
  $[g^0, g^1, g^2, \ldots, g^{p-2}]$ ist Permutation von $[1, 2, \ldots, p - 1]$

```
let {p = 7; g = 3}
in map (`mod` p) $ take (p-1) $ iterate (*g) 1
[1,3,2,6,4,5]
```

- Inferenzregel $\dfrac{g_1 : p_1, \ldots, g_k : p_k}{g : p}$,
  falls $p - 1 = q_1^{e_1} \cdots q_k^{e_k}$ und $\forall i : g^{(p-1)/q_i} \neq 1 \mod p$
- Vaughan Pratt, *Each Prime has a Succinct Certificate*, SIAM J. Comp. 1975
- es folgt $\in$ NP $\cap$ co-NP, aber *not known to be in* P
- Agrawal, Kayal, Saxena: *Primes is in P*, 2004

Inferenz von Typen

- später implementieren wir das, als statische Analyse im Interpreter/Compiler,
  jetzt geben wir nur die Regel an: $\dfrac{f : T_1 \to T_2, x : T_1}{fx : T_2}$

- Bsp. für Verwendung eines Inferenzsystems: Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow, *Associated Types with Class*, POPL 2005

  Absch. 4.2 (Fig. 2) Grundbereich: $\Theta|\Gamma \vdash e : \sigma$

  means that in type environment Γ and instance environment Θ the expression e has type σ

  Bsp. für ein Regelschema: $\dfrac{(v:\sigma) \in \Gamma}{\Theta|\Gamma \vdash v : \sigma}(\textit{var})$

Inferenz von Werten

- Grundbereich: Aussagen $\mathsf{wert}(p, z)$ mit $p \in \texttt{Exp}$, $z \in \mathbb{Z}$

```
data Exp = Const Integer
         | Plus Exp Exp | Times Exp Exp
```

- Axiome: $\mathsf{wert}(\texttt{Const}z, z)$
- Regeln:
  $\dfrac{\mathsf{wert}(X,a), \mathsf{wert}(Y,b)}{\mathsf{wert}(\texttt{Plus}\ X\ Y, a+b)}$, $\dfrac{\mathsf{wert}(X,a), \mathsf{wert}(Y,b)}{\mathsf{wert}(\texttt{Times}\ X\ Y, a\cdot b)}, \ldots$
- das ist syntaxgesteuerte Semantik:

  für jeden Konstruktor von $p \in \texttt{Exp}$ gibt es genau eine Regel mit Konklusion $\mathsf{wert}(p, \ldots)$

Umgebungen (Spezifikation)

- Grundbereich: Aussagen der Form $\mathsf{wert}(E, p, z)$

  (in Umgebung E hat Programm p den Wert z)

  Umgebungen konstruiert aus $\emptyset$ und $E[v := b]$
- Regeln für Operatoren $\dfrac{\mathsf{wert}(E,X,a), \mathsf{wert}(E,Y,b)}{\mathsf{wert}(E, \texttt{Plus}XY, a+b)}, \ldots$
- Regeln für Umgebungen $\dfrac{}{\mathsf{wert}(E[v := b], v, b)}$, $\dfrac{\mathsf{wert}(E, v', b')}{\mathsf{wert}(E[v := b], v', b')}$ für $v \neq v'$
- Regeln für Bindung: $\dfrac{\mathsf{wert}(E,X,b), \mathsf{wert}(E[v := b], Y, c)}{\mathsf{wert}(E, \texttt{let}\ v = X\ \texttt{in}\ Y, c)}$

Umgebungen (Implementierung)

Umgebung ist (partielle) Funktion von Name nach Wert
Realisierungen: `type Env = String -> Integer`
Operationen:

- `empty :: Env` leere Umgebung
- `lookup :: Env -> String -> Integer`
  Notation: $e(x)$
- `extend :: String -> Integer -> Env -> Env`
  Notation: $e[v := z]$

Beispiel

```
lookup (extend  "y" 4 (extend "x" 3 empty)) "x"
```

entspricht $(\emptyset[x := 3][y := 4])x$

Übung

1. Primalitäts-Zertifikate
   - welche von $2, 4, 8$ sind primitive Wurzel mod 101?
   - vollst. Primfaktorzerlegung von 100 angeben
   - ein vollst. Prim-Zertifikat für 101 angeben.
   - bestimmen Sie $2^{(101-1)/5} \mod 101$ von Hand
     Hinweise: 1. das sind *nicht* 20 Multiplikationen,
     2. es wird *nicht* mit riesengroßen Zahlen gerechnet.
2. Geben Sie den vollständigen Ableitungsbaum an für die Auswertung von

   ```
   let {x = 5} in let {y = 7} in x
   ```

Semantische Bereiche

- bisher: Wert eines arithmetischen Ausdrucks ist Zahl.
- jetzt erweitern (Motivation: if-then-else mit richtigem Typ):

```
data Val = ValInt Int
         | ValBool Bool
```

- typische Verarbeitung:

```
value env x = case x of
  Plus l r ->
    case value env l of
      ValInt l ->
        case value env r of
          ValInt r ->
            ValInt ( i + j )
```

Continuations

- Programmablauf-Abstraktion durch Continuations:

```
with_int  :: Val -> (Int  -> Val) -> Val
with_int v k = case v of
    ValInt i -> k i
    _ -> error "expected ValInt"
```

  `k` ist die *continuation* (die Fortsetzung im Erfolgsfall)

- eben geschriebenen Code refaktorisieren zu:

```
value env x = case x of
    Plus l r ->
        with_int ( value env l ) $ \ i ->
        with_int ( value env r ) $ \ j ->
        ValInt ( i + j )
```

Aufgaben

1. Bool im Interpreter
   - Boolesche Literale
   - relationale Operatoren (==, <, o.ä.),
   - Inferenz-Regel(n) für Auswertung des `If`
   - Implementierung der Auswertung von if/then/else mit `with_bool`,
2. Striktheit der Auswertung
   - einen Ausdruck `e :: Exp` angeben, für den `value undefined e` eine Exception ist (zwei mögliche Gründe: nicht gebundene Variable, Laufzeit-Typfehler)
   - mit diesem Ausdruck: diskutiere Auswertung von `let {x = e} in 42`
3. bessere Organisation der Quelltexte
   - Cabalisierung (Quelltexte in `src/`, Projektbeschreibungsdatei `cb.cabal`), Anwendung: `cabal repl` usw.
   - separate Module für `Exp`, `Env`, `Value`,

2 Unterprogramme

Beispiele

- in verschiedenen Prog.-Sprachen gibt es verschiedene Formen von Unterprogrammen:
  - Prozedur, sog. Funktion, Methode, Operator, Delegate, anonymes Unterprogramm
- allgemeinstes Modell:
  - Kalkül der anonymen Funktionen (Lambda-Kalkül),

Interpreter mit Funktionen

- abstrakte Syntax:

```
data Exp = ...
  | Abs { par :: Name , body :: Exp }
  | App { fun :: Exp  , arg :: Exp }
```

- konkrete Syntax:

```
let { f = \ x -> x * x }  in  f (f 3)
```

- konkrete Syntax (Alternative):

```
let { f x = x * x }  in  f (f 3)
```

Semantik (mit Funktionen)

- erweitere den Bereich der Werte:

```
data Val = ... | ValFun ( Val -> Val )
```

- erweitere Interpreter:

```
value :: Env -> Exp -> Val
value env x = case x of
    ... | Abs n b -> _ | App f a -> _
```

- mit Hilfsfunktion `with_fun :: Val -> ...`
- Testfall (in konkreter Syntax)

```
let { x = 4 } in  let { f = \ y -> x * y }
    in  let { x = 5 }  in  f x
```

Let und Lambda

- `let { x = A } in Q`
  kann übersetzt werden in
  `(\ x -> Q) A`
- `let { x = a , y = b } in Q`
  wird übersetzt in ...
- beachte: das ist nicht das `let` aus Haskell

Mehrstellige Funktionen

. . . simulieren durch einstellige:

- mehrstellige Abstraktion:

```
\ x y z -> B := \x -> (\y -> (\z -> B ))
```

- mehrstellige Applikation:

```
f P Q R    :=   ((f P) Q) R
```

  (die Applikation ist links-assoziativ)

- der Typ einer mehrstelligen Funktion:

```
T1 -> T2 -> T3 -> T4   :=
   T1 -> (T2 -> (T3 -> T4))
```

  (der Typ-Pfeil ist rechts-assoziativ)

Semantik mit Closures

- bisher: `ValFun` ist Funktion als Datum der Gastsprache

```
value env x = case x of ...
  Abs n b -> ValFun $ \ v ->
    value (extend n v env) b
  App f a ->
    with_fun ( value env f ) $ \ g ->
    with_val ( value env a ) $ \ v -> g v
```

- alternativ: Closure: enthält Umgebung `env` und Code `b`

```
value env x = case x of ...
  Abs n b -> ValClos env n b
  App f a -> ...
```

Closures (Spezifikation)

- Closure konstruieren (Axiom-Schema):

$$\frac{}{\mathsf{wert}(E, \lambda n.b, \mathsf{Clos}(E, n, b))}$$

- Closure benutzen (Regel-Schema, 3 Prämissen)

$$\frac{\mathsf{wert}(E_1, f, \mathsf{Clos}(E_2, n, b)),\ \mathsf{wert}(E_1, a, w), \mathsf{wert}(E_2[n := w], b, r)}{\mathsf{wert}(E_1, fa, r)}$$

- Ü: Inferenz-Baum für Auswertung des vorigen Testfalls (geschachtelte Let) zeichnen
- ... oder Interpreter so erweitern, daß dieser Baum ausgegeben wird

Rekursion?

- Das geht nicht, und soll auch nicht gehen:

```
let { x = 1 + x } in x
```

- aber das hätten wir doch gern:

```
let { f = \ x -> if x > 0
                 then x * f (x -1) else 1
    } in  f 5
```

(nächste Woche)

- aber auch mit nicht rekursiven Funktionen kann man interessante Programme schreiben:

Testfall (2)

```
let { t f x = f (f x) }
in  let { s x = x + 1 }
    in  t t t t s 0
```

- auf dem Papier den Wert bestimmen

- mit selbstgebautem Interpreter ausrechnen
- mit Haskell ausrechnen
- in JS (node) ausrechnen

Repräsentation von Fehlern

- Fehler explizit im semantischen Bereich des Interpreters repräsentieren (anstatt als Exception der Gastsprache)

```
data Val = ... | ValErr Text
```

- strikte Semantik: `ValErr` *niemals* in Umgebung (bei Let-Bindung oder UP-Aufruf)
- Ü: realisieren durch Aufruf (an geeigneten Stellen) von

```
with_val :: Val -> (Val -> Val) -> Val
with_val v k = case v of
  ValErr _ -> v
  _ -> k v
```

Übungen

1. eingebaute primitive Rekursion (Induktion über Peano-Zahlen):

   implementieren Sie die Funktion `fold :: r -> (r -> r) -> N -> r`

   Testfall: `fold 1 (\x -> 2*x) 5 == 32`

   durch `data Exp = .. | Fold ..` und neuen Zweig in `value`

   Wie kann man damit die Fakultät implementieren?
2. alternative Implementierung von Umgebungen
   - bisher `type Env = Id -> Val`
   - jetzt `type Env = Data.Map.Map Id Val` oder `Data.HashMap`

   Messung der Auswirkungen: 1. Laufzeit eines Testfalls, 2. Laufzeiten einzelner UP-Aufrufe (profiling)

3 Lambda-Kalkül (Wdhlg.)

Motivation

1. Modellierung von Funktionen:

- intensional: Fkt. ist Berechnungsvorschrift, Programm
- (extensional: Fkt. ist Menge v. geordneten Paaren)

2. Notation mit gebundenen (lokalen) Variablen, wie in

- Analysis: $\int x^2 \,\mathrm{d}\, x, \sum_{k=0}^{n} k^2$
- Logik: $\forall x \in A : \forall y \in B : P(x, y)$
- Programmierung: `static int foo (int x) { ... }`

Der Lambda-Kalkül

- ist der Kalkül für Funktionen mit benannten Variablen
- Alonzo Church, 1936 ... Henk Barendregt, 1984 ...
- die wesentliche Operation ist das Anwenden einer Funktion:

$$(\lambda x.B)A \to_\beta B[x := A]$$

  Beispiel: $(\lambda x.x * x)(3 + 2) \to_\beta (3 + 2) * (3 + 2)$

- Im reinen Lambda-Kalkül gibt es *nur* Funktionen
  (keine Zahlen, Wahrheitswerte usw.)

Lambda-Terme

- Menge Λ der Lambda-Terme
  (mit Variablen aus einer Menge V):
  - (Variable) wenn $x \in V$, dann $x \in \Lambda$
  - (Applikation) wenn $F \in \Lambda, A \in \Lambda$, dann $(FA) \in \Lambda$
  - (Abstraktion) wenn $x \in V, B \in \Lambda$, dann $(\lambda x.B) \in \Lambda$

Beispiele: $x, (\lambda x.x), ((xz)(yz)), (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

- verkürzte Notation (Klammern weglassen)
  - $(\dots((FA_1)A_2)\dots A_n) \sim FA_1A_2\dots A_n$
  - $\lambda x_1.(\lambda x_2.\dots.(\lambda x_n.B)\dots) \sim \lambda x_1x_2\dots x_n.B$

  mit diesen Abkürzungen simuliert $(\lambda x_1\dots x_n.B)A_1\dots A_n$ eine mehrstellige Funktion und -Anwendung

Eigenschaften der Reduktion

- $\rightarrow_\beta$ auf Λ ist nicht terminierend (es gibt Terme mit unendlichen Ableitungen)
  $W = \lambda x.xx, \Omega = WW.$
- es gibt Terme mit Normalform und unendlichen Ableitungen, $KI\Omega$ mit $K = \lambda xy.x, I = \lambda x.x$
- $\rightarrow_\beta$ auf Λ ist konfluent
  $\forall A, B, C \in \Lambda : A \rightarrow_\beta^* B \wedge A \rightarrow_\beta^* C \Rightarrow \exists D \in \Lambda : B \rightarrow_\beta^* D \wedge C \rightarrow_\beta^* D$
- Folgerung: jeder Term hat höchstens eine Normalform

Beziehung zur Semantik des Interpreters

- λ-Kalkül: Rel. $\rightarrow_\beta$ substituiert Variablen im Term

  schwache Reduktion: wie $\rightarrow_\beta$, aber niemals unter λ

  unser Interpreter: realisiert schwache Reduktion, Regeln für $\mathsf{wert}(E, X, w)$ speichern Substitutionen in Umgebung
- ein Zusammenhang wird hergestellt durch *Kalküle für explizite Substitutionen*,

  Pierre-Louis Curien: *An Abstract Framework for Environment Machines*, TCS 82 (1991), `https://doi.org/10.1016/0304-3975(91)90230-Y`

  Abadi, Cardelli, Curien, Levy: *Explicit Substitutions*, JFP 1991, `https://doi.org/10.1017/S0956796800000186`,

Daten als Funktionen

- Simulation von Daten (Tupel) durch Funktionen (Lambda-Ausdrücke):
  - Konstruktor: $\langle D_1, \ldots, D_k \rangle := \lambda s.sD_1 \ldots D_k$
  - Selektoren: $s_i^k := \lambda t.t(\lambda d_1 \ldots d_k.d_i)$

  es gilt $s_i^k \langle D_1, \ldots, D_k \rangle \rightarrow_\beta^* D_i$

  Ü: überprüfen für $k = 2$
- Anwendungen:
  - Modellierung von Listen, Zahlen
  - Auflösung simultaner Rekursion

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:
  $\mathsf{True} := \lambda xy.x, \mathsf{False} := \lambda xy.y$
- Verzweigung: $\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y := bxy$
- natürliche Zahlen als iterierte Paare (Ansatz)
  $(0) := \langle \mathsf{True}, \lambda x.x \rangle;\ (n+1) := \langle \mathsf{False}, n \rangle$
- s_2^2 ist partielle Vorgänger-Funktion: $s_2^2(n+1) = n$
- Verzweigung: $\texttt{if}\ a = 0\ \texttt{then}\ x\ \texttt{else}\ y := s_1^2 axy$
- Ü: nachrechnen. Ü: das geht sogar mit $(0) = \lambda x.x$
- Rekursion?

Fixpunkt-Kombinatoren (Motivation)

- Beispiel: die Fakultät

```
f = \ x -> if x=0 then 1 else x*f(x-1)
```

  erhalten wir als Fixpunkt einer Fkt. 2. Ordnung

```
g = \ h x -> if x=0 then 1 else x * h(x-1)
f = fix g   -- d.h., f = g f
```

- Ü: $g(\lambda z.z)7$, Ü: `fix` $g\ 7$
- Implementierung von `fix` mit Rekursion:

  ```
  fix g = g (fix g)
  ```

- es geht aber auch *ohne Rekursion*. Ansatz: `fix` $= AA$,
  dann `fix` $g = AAg = g(AAg) = g($`fix` $g)$
  eine Lösung ist $A = \lambda xy. \ldots$

Fixpunkt-Kombinatoren (Implementierung)

- Definition (der *Fixpunkt-Kombinator* von Turing)
  $\Theta = (\lambda xy.(y(xxy)))(\lambda xy.(y(xxy)))$
- Satz: $\Theta f \rightarrow_\beta^* f(\Theta f)$, d. h. Θf ist Fixpunkt von f
- Folgerung: im Lambda-Kalkül kann man simulieren:
  - Daten: Zahlen, Tupel von Zahlen
  - Programmablaufsteuerung durch:
    - Nacheinander„ausführung“:
      Verkettung von Funktionen
    - Verzweigung,
    - Wiederholung: durch Rekursion (mit Fixpunktkomb.)

Lambda-Berechenbarkeit

Satz: (Church, Turing)

Menge der Turing-berechenbaren Funktionen
(Zahlen als Wörter auf Band)
Alan Turing: On Computable Numbers, with an Application to the Entscheidungsproblem, Proc. LMS, 2 (1937) 42 (1) 230–265 `https://dx.doi.org/10.1112/plms/s2-42.1.230`

$=$ Menge der Lambda-berechenbaren Funktionen
(Zahlen als Lambda-Ausdrücke)
Alonzo Church: A Note on the Entscheidungsproblem, J. Symbolic Logic 1 (1936) 1, 40–41

$=$ Menge der while-berechenbaren Funktionen
(Zahlen als Registerinhalte)

Kodierung von Zahlen nach Church

- $c : \mathbb{N} \to \Lambda : n \mapsto \lambda fx.f^n(x)$
  mit $f^0(x) := x$, $f^{n+1}(x) := f(f^n(x))$
- in Haskell: `c n f x = iterate f x !! n`
- Decodierung: `d e = e (\x -> x+1) 0`
- Nachfolger: $s(c_n) = c_{n+1}$ für $s = \lambda nfx.f(nfx)$
  1. auf Papier beweisen, 2. mit leancheck prüfen
  benutze `check $ \ (Natural x) -> ...`
- Addition: `plus` c_a $c_b = c_{a+b}$ für `plus` $= \lambda abfx.af(bfx)$
- implementiere die Multiplikation, beweise, prüfe
- Potenz: `pow` c_a $c_b = c_{a^b}$ für `pow` $= \lambda ab.ba$

Übung Lambda-Kalkül (I)

- die Fakultät (z.B. von 7) ...
  - in Haskell (ohne Rekursion, aber mit `Data.Function.fix`)
  - in unserem Interpreter (ohne Rekursion, mit Turing-Fixpunktkombinator Θ)
  - in Javascript (ohne Rekursion, mit Θ)
- Kodierung von Wahrheitswerten und Zahlen (nach Church)
  - implementiere Test auf 0: `iszero` c_n = if $n = 0$ then True else False
  - implementiere Addition, Multiplikation, Fakultät
    ohne `If`, `Eq`, `Const`, `Plus`, `Times`
  - für nützliche Ausgaben: das Resultat nach `ValInt` dekodieren (dabei muß `Plus` und `Const` benutzt werden)

Übung Lambda-Kalkül (II)

folgende Aufgaben aus H. Barendregt: *Lambda Calculus*

- (Abschn. 6.1.5) gesucht wird F mit $Fxy = FyxF$.
  Musterlösung: es gilt $F = \lambda xy.FyxF = (\lambda fxy.fyxf)F$,
  also $F = \Theta(\lambda fxy.fyxf)$

- (Aufg. 6.8.2) Konstruiere $K^\infty \in \Lambda^0$ (ohne freie Variablen) mit $K^\infty x = K^\infty$ (hier und in im folgenden hat $=$ die Bedeutung $(\rightarrow_\beta \cup \rightarrow_\beta^-)^*$
- Konstruiere $A \in \Lambda^0$ mit $Ax = xA$
- beweise den Doppelfixpunktsatz (Kap. 6.5)

  $\forall F, G : \exists A, B : A = FAB \wedge B = GAB$
- (Aufg. 6.8.17, B. Friedman) Konstruiere Null, Nulltest, partielle Vorgängerfunktion für Zahlensystem mit Nachfolgerfunktion $s = \lambda x.\langle x \rangle$ (das 1-Tupel)
- (Aufg. 6.8.14, J. W. Klop)

$$\begin{aligned} X &= \lambda abcdefghijklmnopqstuvvxyzr. \\ &\quad r(thisisafixedpointcombinator) \\ Y &= X^{27} = \underbrace{X \ldots X}_{27} \end{aligned}$$

  Zeige, daß Y ein Fixpunktkombinator ist.

4 Fixpunkte

Motivation

Das ging bisher gar nicht:

```
let { f = \ x -> if x > 0
                 then x * f (x -1) else 1
    } in  f 5
```

Lösung 1: benutze Fixpunktkombinator

```
let { Theta =  ... } in
let { f = Theta ( \ g -> \ x -> if x > 0
                 then x * g (x - 1) else 1 )
    } in f 5
```

Lösung 2 (später): realisiere Fixpunktberechnung im Interpreter (neuer AST-Knotentyp `Fix`)

Existenz von Fixpunkten

Fixpunkt von $f :: C \to C$ ist $x :: C$ mit $fx = x$.

Existenz? Eindeutigkeit? Konstruktion?

Satz: Wenn C *pointed CPO* und f *stetig*, dann besitzt f genau einen kleinsten Fixpunkt.

- CPO = complete partial order = vollständige Halbordnung
- complete = jede monotone Folge besitzt Supremum (= kleinste obere Schranke)
- pointed: C hat kleinstes Element $\bot$

Ü (Wdhlg) Def. obere Schranke, Supremum

Beispiele f. Halbordnungen, CPOs

Halbordnung? pointed? complete?

- $\leq$ auf $\mathbb{N}$
- $\leq$ auf $\mathbb{N} \cup \{+\infty\}$
- $\leq$ auf $\{x \mid x \in \mathbb{R}, 0 \leq x \leq 1\}$
- $\leq$ auf $\{x \mid x \in \mathbb{Q}, 0 \leq x \leq 1\}$
- Teilbarkeit auf $\mathbb{N}$
- Präfix-Relation auf Σ^*
- $\{((x_1, y_1), (x_2, y_2)) \mid (x_1 \leq x_2) \vee (y_1 \leq y_2)\}$ auf $\mathbb{R}^2$
- $\{((x_1, y_1), (x_2, y_2)) \mid (x_1 \leq x_2) \wedge (y_1 \leq y_2)\}$ auf $\mathbb{R}^2$
- Relation $\subseteq$ auf $\{\{A\}, \{B\}, \{A, B\}\}$
- identische Relation id_M auf einer beliebigen Menge M
- $\{(\bot, x) \mid x \in M_\bot\} \cup \mathrm{id}_M$ auf $M_\bot := \{\bot\} \cup M$

Stetige Funktionen

f ist stetig :=

- f ist monoton: $x \leq y \Rightarrow f(x) \leq f(y)$
- und für monotone Folgen $[x_0, x_1, \ldots]$ gilt: $f(\sup[x_0, x_1, \ldots]) = \sup[f(x_0), f(x_1), \ldots]$

Beispiele: in $(\mathbb{N} \cup \{+\infty\}, \leq)$

- $x \mapsto 42$ ist stetig

- $x \mapsto$ if $x < +\infty$ then $x + 1$ else $+\infty$
- $x \mapsto$ if $x < +\infty$ then 42 else $+\infty$

Satz: Wenn C *pointed CPO* und $f : C \to C$ *stetig*, dann besitzt f genau einen kleinsten Fixpunkt ...

Beweis: ... und dieser ist $\sup[\bot, f(\bot), f^2(\bot), \ldots]$

CPO auf Menge von Funktionen

- Menge der partiellen Funktionen von B nach B:
  $C = (B \hookrightarrow B)$
- partielle Funktion $f : B \hookrightarrow B$ entspricht totaler Funktion $f : B \to B_\bot$
- C geordnet durch $f \leq g \iff \forall x \in B : f(x) \leq g(x)$,
  wobei $\leq$ die vorhin definierte CPO auf $B_\bot$
- $f \leq g$ bedeutet: g ist Verfeinerung von f
- Das Bottom-Element von C ist die überall undefinierte Funktion. (diese heißt auch $\bot$)

Funktionen als CPO, Beispiel

- der Operator $F =$

```
\ g -> ( \ x -> if (x==0) then 0
                else 2 + g (x - 1) )
```

  ist stetig auf $(\mathbb{N} \hookrightarrow \mathbb{N})$ (Beispiele nachrechnen!)
- Iterative Berechnung des Fixpunktes:

$$\begin{aligned} \bot &= \emptyset \quad \text{überall undefiniert} \\ F\bot &= \{(0,0)\} \\ F(F\bot) &= \{(0,0),(1,2)\} \\ F^3\bot &= \{(0,0),(1,2),(2,4)\} \end{aligned}$$

- $\sup[\ldots, F^k\bot, \ldots] = \{(x, 2x) \mid x \in \mathbb{N}\}$

Welche Funktionale sind stetig?

- die Erfahrung (der Programmierung mit rekursiven Funktionen) lehrt: alle Operatoren `\g -> \x -> .. g ..` sind stetig.
- denn die Semantik der (von uns bisher benutzten) AST-Konstruktoren bildet stetige Ftk. auf stetige Fkt. ab
- Ü: $(\lambda x.\, \mathsf{if}\, x = \bot\, \mathsf{then}\, 42\, \mathsf{else}\, \bot)$ ist nicht stetig.
- Diskussion: eine solche Funktion kann nicht die Semantik eines Programms sein, weil das Halteproblem nicht entscheidbar ist

Fixpunkte und Laziness

- Fixpunkte existieren in pointed CPOs.
  - (Maschinen-) Zahlen: nicht pointed
    (arithmetische Operatoren sind strikt)
  - (Haskell-) Daten (Listen, Bäume usw.): pointed:
    (Konstruktoren sind nicht strikt)
  - Funktionen (Unterprogramme):
    Abstraktion (λ) ist nicht strikt, in *jeder* Sprache!
- Beispiele in Haskell:

```
fix f = f (fix f)
xs = fix $ \ zs -> 1 : zs
g = fix $ \h -> \x -> if x==0 then 1 else x*h(x-1)
```

- in Sprachen mit strikter Semantik üblicherweise:
  keine rekursiven Daten, aber rekursive Unterprogramme

Fixpunktberechnung im Interpreter

- Erweiterung der abstrakten Syntax:

```
data Exp = ... | Rec Name Exp
```

- Beispiel

```
App
  (Rec g (Abs v (if v==0 then 0 else 2 + g(v-1))))
  5
```

- Bedeutung: `Rec x B` ist Fixpunkt von $(\lambda x.B)$
- Semantik: $\dfrac{\mathsf{wert}(E, (\lambda x.B)(\mathsf{Rec}\ x\ B), v)}{\mathsf{wert}(E, \mathsf{Rec}\ x\ B, v)}$
- Ü: verwende `Let` statt `App (Abs ..) ..`
- Ü: das Beispiel mit dieser Regel auswerten

Arithmetik mit Bedarfsauswertung

- über $\mathbb{Q}$ hat $f = \lambda x.1 + x/4$ einen Fixpunkt $(4/3)$,
  aber $\sup_k f^k \bot = \bot$, weil die Operationen strikt sind.
- wirklich? Kommt auf die Repräsentation der Zahlen an!
  Op. auf Maschinenzahlen sind strikt. — Aber:
- Zahl als lazy Liste von Ziffern (Bsp: Basis 2)
  `x=plus(1:repeat 0)(0:0:x)=[1,0,1,0,1,0..]`
- Ü: bestimme $y = \sqrt{2} - 1$ aus $2 = (1 + y)^2$,
  d.h., als Fixpunkt von $\lambda y.(1 - y^2)/2$
- Kombiniere `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.4249` (Jerzy Karczmarczuk 1998), `http://joerg.endrullis.de/publications/productivity/`

Simultane Rekursion: letrec

- Beispiel (aus: D. Hofstadter, Gödel Escher Bach, 1979)

```
letrec { f = \ x -> if x == 0 then 1
                    else x - g(f(x-1))
       , g = \ x -> if x == 0 then 0
                    else x - f(g(x-1))
} in f 15
```

  Bastelaufgabe: für welche x gilt $f(x) \neq g(x)$?

- weitere Beispiele:

```
letrec { y = x * x, x = 3 + 4  } in x - y
letrec { f = \ x -> .. f (x-1) } in f 3
```

letrec nach rec (falsch)

- Plan: mit Lambda-Ausdrücken für Konstruktor, Selektor

```
LetRec [(n1,x1), .. (nk,xk)] y
=> ( rec t ( let n1 = select1 t
                  ...
                  nk = selectk t
             in  tuple x1 .. xk ) )
   ( \ n1 .. nk -> y )
```

- benutzt $\langle x_1, \ldots, x_k \rangle f = f x_1 \ldots x_k$
- terminiert nicht, die Auswertungsstrategie des Interpreters ist dafür zu eifrig (*eager*)
- Lösung: `tuple` direkt unter `rec t`,
  `let .. tuple ..` ⇒ `tuple (let ..) (let ..)`

letrec nach rec (richtig)

- Teilausdrücke (für jedes i)

```
let { n1 = select1 t, ..  nk = selectk t
    } in xi
```

  äquivalent vereinfachen zu `t (\ n1 .. nk -> xi)`

- ```
  LetRec [(n1,x1), .. (nk,xk)] y
  => ( rec t
       ( tuple ( t ( \ n1 .. nk -> x1 ))
               ...
               ( t ( \ n1 .. nk -> xk )) ))
     ( \ n1 .. nk -> y )
  ```

- Ü: implementiere `letrec {f = _, g = _} in f 15`
  ```

Übung Fixpunkte

1. Limes der Folge $F^k(\bot)$ für

```
F h = \ x -> if x > 23 then x - 11
              else h (h (x + 14))
```

2. Ist F stetig? Gib den kleinsten Fixpunkt von F an:

```
F h =  \ x -> if x >= 2 then 1 + h(x-2)
    else if x == 1 then 1 else h(4) - 2
```

   Hat F weitere Fixpunkte?

3. $C =$ Menge der Formalen Sprachen über $\Sigma = \{a, b\}$, halbgeordnet durch $\subseteq$. ist CPO? pointed?
   $g : C \to C : L \mapsto \{\epsilon\} \cup \{a\} \cdot L \cdot \{b\}$ ist stetig? Fixpunkt(e)?
   $h : C \to C : L \mapsto \{a\} \cup L \cdot \{b\} \cdot (\Sigma^* \setminus L)$

4. in der Relation $\subseteq$ auf $\{\{A\}, \{B\}, \{A, B\}\}$: geben Sie eine stetige Funktion an, die zwei verschiedene kleinste Fixpunkte besitzt.
   Ist das eine Widerspruch zum CPO-Theorem?

5. Geben Sie Argumente aus dieser Diskussion wieder: ... *distinguish bindings that are self-referentially recursive from non-recursive bindings* `https://github.com/ghc-proposals/ghc-proposals/pull/401` (O. Charles, 8. Febr. 2021)
   Vergleichen Sie mit `let (rec)` in OCaml (Primärquelle angeben, d.h., Sprachstandard)
   Für Haskell-Lückentext-Aufgaben in autotool: um die Benutzung von Rekursionsmustern zu erzwingen, wäre es nützlich, beliebige Rekursion zu verbieten.