Logische Methoden des Software Engineerings

Prof. Dr. Jakob Rehof

Lehrstuhl 14, Software Engineering

λ_Δ-calculus [Rehof & Sørensen, 1994]

8.3.2. DEFINITION (type-free and simply typed λ_Δ-calculus, $\lambda_\Delta(\to)$). The term language Λ_Δ of type-free λ_Δ-calculus is defined by the grammar:

$$\Lambda_\Delta \ni M ::= x \mid \lambda x.M \mid M\ M' \mid \Delta x.M$$

The simply typed λ_Δ-calculus has as type language the set $L(\to)$. The inference system of simply typed λ_Δ-calculus arises from the system of simply typed λ-calculus à la Curry by addition of the rule

$$\frac{\Gamma, x : \varphi \to \bot \vdash M : \bot}{\Gamma \vdash \Delta x.M : \varphi}$$

A new binding operator to assign proof term to applications of (--E) rule

Classical proof normalization

8.4.1. DEFINITION. Define the relation $\to_\Delta$ on Λ_Δ à la Church as the smallest compatible relation containing the following rules:

$$
\begin{array}{lcll}
(\Delta x{:}\neg(\varphi \to \psi).M)\ N & \to_\Delta & \Delta z{:}\neg\psi.M\{x := \lambda y{:}\varphi \to \psi\ .\ z\ (y\ N)\}; \\
\Delta x{:}\neg\varphi.x\ M & \to_\Delta & M & \text{provided } x \notin \mathrm{FV}(M); \\
\Delta x{:}\neg\varphi.x\ \Delta y{:}\neg\varphi.N & \to_\Delta & \Delta z{:}\neg\varphi.N\{x, y := z\}.
\end{array}
$$

From the point of view of reduction on proofs, the first of these rules decreases the complexity of formulas to which we apply the double negation elimination rule. The second rule may be regarded as a form of η-rule for Δ. The third rule reduces applications of the double negation elimination rule nested in a certain trivial way.

Pairs and sums

8.5.2. DEFINITION. Define the following abbreviations.

$$
\begin{array}{lcl}
< P, Q > & = & \lambda z.z\ P\ Q; \\
\pi_i(P) & = & \Delta k.P\ (\lambda x_1.\lambda x_2.k\ x_i); \\
\mathrm{in}_i(P) & = & \lambda y_1.\lambda y_2.y_i\ P; \\
\mathrm{case}(P; x_1.Q_1; x_2.Q_2) & = & \Delta k.P\ (\lambda x_1.kQ_1)\ (\lambda x_2.kQ_2).
\end{array}
$$

Projection

$< P, Q >$	$=$	$\lambda z.z\ P\ Q;$
$\pi_i(P)$	$=$	$\Delta k.P\ (\lambda x_1.\lambda x_2.k\ x_i);$

The definition of pairs is the standard one from type-free λ-calculus (see Chapter 1), while the projection construction is different from that normally employed in type-free λ-calculus, viz. $M\ (\lambda x_1.\lambda x_2.x_i)$. This latter definition does not work because $\lambda x_1.\lambda x_2.x_i$ has type $\varphi_1 \to \varphi_2 \to \varphi_i$ instead of the type $\varphi_1 \to \varphi_2 \to \bot$, which M expects. Changing the definition of conjunctive types to solve the problem is not possible; it leads to the type of a pair being dependent on which component a surrounding projection picks.[1] The operator Δ solves the problem by means of an application which turns the type of x_i into $\bot$ regardless of i. When the projection is calculated, the k reaches its Δ and can be removed by the second reduction rule for Δ:

$$
\begin{array}{rcl}
\pi_1(< M_1, M_2 >) & \equiv & \Delta k.(\lambda f.f\ M_1\ M_2)\ \lambda x_1.\lambda x_2.k\ x_1 \\
& \to_\beta & \Delta k.(\lambda x_1.\lambda x_2.k\ x_1)\ M_1\ M_2 \\
& \twoheadrightarrow_\beta & \Delta k.k\ M_1 \\
& \to_\Delta & M_1
\end{array}
$$

λ_Δ- calculus from [R&S 1994]

Note (JR)

λ_Δ-calculus from [R& S 1994]

Write $\nabla(M) \equiv \Delta d.M$ with $d \notin FV(M)$.

$$
\begin{array}{llll}
(\Delta x.M)\ N & \longrightarrow_\Delta & \Delta z.M\{x := \lambda y.z\ (y\ N)\} & (\textit{Reduction}) \\
\Delta x.x\ M & \longrightarrow_\Delta & M, \text{ provided } x \notin FV(M) & (\Delta\textit{-elimination}) \\
\Delta x.x\ \nabla(x\ M) & \longrightarrow_\Delta & M, \text{ provided } x \notin FV(M) & (\Delta\textit{-elimination})
\end{array}
$$

Rules from LCHI :

8.4.1. DEFINITION. Define the relation $\to_\Delta$ on Λ_Δ à la Church as the smallest compatible relation containing the following rules:

$$
\begin{array}{lll}
(\Delta x{:}\neg(\varphi \to \psi).M)\ N & \to_\Delta & \Delta z{:}\neg\psi.M\{x := \lambda y{:}\varphi \to \psi\ .\ z\ (y\ N)\}; \\
\Delta x{:}\neg\varphi.x\ M & \to_\Delta & M \qquad \text{provided } x \notin \mathrm{FV}(M); \\
\Delta x{:}\neg\varphi.x\ \Delta y{:}\neg\varphi.N & \to_\Delta & \Delta z{:}\neg\varphi.N\{x, y := z\}.
\end{array}
$$

λ_Δ- calculus from [R&S 1994]

Note (JR)

λ_Δ-calculus from [R&S 1994]

The second Δ-reduction rule of LCHI is more general, having the second Δ-elimination rule of [R&S 1994] as a special case. For we have, assuming $x \notin FV(M)$ and writing momentarily $\Longrightarrow_\Delta$ for the Δ-rules of LCHI:

$$\begin{array}{rcl} \Delta x.x\ \nabla(x\ M) & \equiv & \Delta x.x\ \Delta d.x\ M \\ & \Longrightarrow_\Delta & \Delta z.(x\ M)\{x, d := z\} \\ & \equiv & \Delta z.z\ M \\ & \Longrightarrow_\Delta & M \end{array}$$

Rules from LCHI :

8.4.1. DEFINITION. Define the relation $\to_\Delta$ on Λ_Δ à la Church as the smallest compatible relation containing the following rules:

$$\begin{array}{lcll} (\Delta x{:}\neg(\varphi \to \psi).M)\ N & \to_\Delta & \Delta z{:}\neg\psi.M\{x := \lambda y{:}\varphi \to \psi\ .\ z\ (y\ N)\}; \\ \Delta x{:}\neg\varphi.x\ M & \to_\Delta & M & \text{provided } x \notin \mathrm{FV}(M); \\ \Delta x{:}\neg\varphi.x\ \Delta y{:}\neg\varphi.N & \to_\Delta & \Delta z{:}\neg\varphi.N\{x, y := z\}. \end{array}$$

λ_Δ- calculus from [R&S 1994]

Common Lisp [Ste84] programmers may read $\Delta x.x\ M$ as `catch` x `in` M and $\nabla(x\ N)$ as `throw` $x\ N$. ML [Mil90] programmers may read $\Delta x.x\ M$ as M `handle` $X(v)$ `=>` v and $\nabla(x\ N)$ as `raise` $X(N)$ where X is declared `exception` X `of` $\mathcal{P}$, $\mathcal{P}$ being the intented type of N. Readers familiar with Felleisen's control operators [Fel87b] may think of $\Delta x.M$ and $\nabla(K)$ as $\mathcal{F}(\lambda x.M)$ and $\mathcal{A}(K)$, respectively, see Section 4.

Definition 2. (Reduction on constructions.)

(1a)	$(\lambda x.M)\ N$	$\rightarrow M\{x := N\}$
(1b)	$\pi_i(< M_1, M_2 >)$	$\rightarrow M_i$
(1c)	$\mathrm{case}(\mathrm{in}_i(L); x_1.M_1; x_2.M_2)$	$\rightarrow M_i\{x_i := L\}$
(2a)	$(\Delta x.M)\ N$	$\rightarrow \Delta z.M\{x := \lambda y.z\ (y\ N)\}$
(2b)	$\pi_i(\Delta x.M)$	$\rightarrow \Delta u.M\{x := \lambda y.u\ \pi_i(y)\}$
(2c)	$case(\Delta x.M; y_1.M_1; y_2.M_2)$	$\rightarrow \Delta u.M\{x := \lambda y.u\ case(y; y_1.M_1; y_2.M_2)\}$
(3)	$\Delta x.x\ M$	$\rightarrow M$ provided $x \notin FV(M)$
(4)	$\Delta x.x\ \nabla(x\ M)$	$\rightarrow M$ provided $x \notin FV(M)$

These rules are the reduction rules for $\lambda_{\Delta P}$. Omitting (1b),(1c),(2b),(2c) yields the reduction rules for λ_Δ.

The construction $\nabla(M)$ is an abbreviation for $\Delta d.M$ where $d \notin FV(M)$.

Factorization of $\mathcal{F}$

We can factor the operational semantics of the control operator into a single context-dependent (*computation*) rule, which applies only at the top level (i.e., in the empty context) and local (*reduction*) rules which can be applied independently of context:

$$(\mathcal{F}_T)\ \mathcal{F}(M) \to M\ (\lambda x.x)$$

Top-level rule: applicable only in empty context

$$(\mathcal{F}_L)\ \mathcal{F}(M)\ N \to \mathcal{F}(\lambda\kappa.M(\lambda f.\kappa\ (f\ N)))$$

$$(\mathcal{F}_R)\ M\ \mathcal{F}(N) \to \mathcal{F}(\lambda\kappa.N(\lambda f.\kappa\ (M\ f)))\ ,\quad \text{provided } M \text{ is a value}$$

$\mathcal{F}$ and Δ

$$
\begin{array}{rcl}
\underline{(\Delta x.M)\ N} & \equiv & \mathcal{F}(\lambda x.\underline{M})\ \underline{N} \\
& \rhd_{\mathcal{F}_L} & \mathcal{F}(\lambda\kappa.(\lambda x.\underline{M})(\lambda f.\kappa\ (f\ \underline{N}))) \\
& \rhd_{\beta} & \mathcal{F}(\lambda\kappa.\underline{M}\{x := \lambda f.\kappa\ (f\ \underline{N})\}) \\
& \equiv & \underline{\Delta\kappa.M\{x := \lambda f.\kappa\ (f\ N)\}}
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\Delta x.x\ M} & \equiv & \mathcal{F}(\lambda x.x\ \underline{M}) \\
& \rhd_{\mathcal{F}_T} & (\lambda x.x\ \underline{M})(\lambda y.y) \\
& \rhd_{\beta}^{*} & \underline{M}
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\Delta x.x\ \nabla(x\ M)} & \equiv & \mathcal{F}(\lambda x.x\ \mathcal{A}(x\ \underline{M})) \\
& \rhd_{\mathcal{F}_T} & (\lambda x.x\ \mathcal{A}(x\ \underline{M}))(\lambda y.y) \\
& \rhd_{\beta}^{*} & \mathcal{A}(\underline{M}) \\
& \rhd_{\mathcal{A}} & \underline{M}
\end{array}
$$

The excluded middle

Note (JR)

As one would expect, we can derive the principle $A \vee \neg A$ in the λ_Δ-calculus. Here is the proof term of that derivation:

$$\vdash \Delta j^{\neg(A \vee \neg A)}.\, j\ (\mathtt{in}_2\ (\lambda x^A.\, j\ (\mathtt{in}_1\ x))) : A \vee \neg A$$

Embedding of classical logic into intuitionistic logic

- A simple (effective) transformation **k** on formulae such that F is provable in CL iff **k**(F) is provable in IL
- Discovered in the 1930‘s (Gödel, Gentzen, Kolmogorov, Kuroda)
- This may surprise, because CL is a stronger logic than IL

Double-negation translation **k**

8.6.1. DEFINITION. Define the translation k from implicational formulas to implicational formulas by:

$$\begin{aligned} k(\alpha) &= \neg\neg\alpha \\ k(\bot) &= \neg\neg\bot \\ k(\varphi \to \psi) &= \neg\neg(k(\varphi) \to k(\psi)) \end{aligned}$$

Embedding

- We will prove the embedding result by defining an accompanying transformation t of proof terms such that:

  – *whenever M is a proof of F in CL,*
    then ***t****(M) is a proof of* ***k****(F) in IL*

Proof transformation **t**

8.6.2. DEFINITION. Define the translation t from λ_Δ-terms to λ-terms by:

$$
\begin{array}{lcl}
t(x) & = & \lambda k.x\ k \\
t(\lambda x.M) & = & \lambda k.k\ \lambda x.t(M) \\
t(M\ N) & = & \lambda k.t(M)\ (\lambda m.m\ t(N)\ k) \\
t(\Delta x.M) & = & \lambda k.(\lambda x.t(M))\ (\lambda h.h\ \lambda j.\lambda i.i\ (j\ k))\ \lambda z.z
\end{array}
$$

Embedding

8.6.3. PROPOSITION. *If* $\Gamma \vdash M : \varphi$ *in simply typed* λ_Δ*-calculus, then* $k(\Gamma) \vdash t(M) : k(\varphi)$ *in simply typed* λ*-calculus.*

This gives another proof of consistency of classical propositional logic: if classical logic is inconsistent, so is intuitionistic logic.

8.6.4. PROPOSITION. $\not\vdash \bot$ *in classical propositional logic.*

PROOF. If $\vdash M : \bot$ in classical propositional logic, then $\vdash t(M) : \neg\neg\bot$ in intuitionistic logic, and then $\vdash t(M)\ \lambda z.z : \bot$ in intuitionistic logic, a contradiction. $\square$

Conservativity of CL over IL

The above proof gives a conservativity result: if $\bot$ is provable in classical logic, $\bot$ is provable already in intuitionistic logic. The construction can be generalized to other formulas than $\bot$; in fact, this way one can prove that any formula of form $\forall x \exists y : P(x, y)$, where P is a primitive recursive predicate, is provable in classical arithmetic (i.e., Peano Arithmetic) iff it is provable in intuitionistic arithmetic (i.e., Heyting Arithmetic). Formulas of this form are quite important since they include, e.g., every assertion that some algorithm terminates ("for any input x there is a terminating computation y"). In other words, as concerns provability of termination of algorithms there is no difference between intuitionistic and classical logic.

Semantic embedding property

8.6.5. PROPOSITION. *If* $M =_{\beta\Delta} N$ *then* $t(M) =_{\beta} t(N)$.

This suggests that something much deeper is going on than just a syntactic game.

Indeed, we shall see that Δ is a control operator and **t** is a CPS-transformation!

Control operators

- Developed in programming language research in semantics, independently of logical considerations
- First appeared in functional languages (but later became mainstream)
  - SCHEME: **call/cc**
  - ML: **exceptions**
  - LISP: **catch/throw**

Example: purely functional tree multiplier

Our first solution is the straightforward purely functional one which trades efficiency off for elegance. We suppose given auxiliary functions `mt?` testing for the empty tree, `num` selecting the node value of the root node, `lson` and `rson` returning the left and right subtrees.

8.7.1. EXAMPLE. (Functional, elegant, inefficient version)

```
M1 = λ t.(if (mt?  t)
        1
        (* (num t) (* (M1 (lson t)))(M1 (rson t)))))
```

Optimized functional version

One can optimize `M1` so as to stop multiplying as soon as a node value of 0 is encountered. This can be done in purely functional style, by means of tests. Our next solution embodies that strategy. Here we assume a constructor `EX` and a test function `EX?` such that `(EX` M`)` tags M with the marker `EX`, and `EX?` tests for the presence of the tag. Furthermore, we assume a 0 test function `zero?`. The reader will probably agree that elegance (or at the least conciseness) has now been traded off for efficiency.

Optimized functional version

8.7.2. EXAMPLE. (Functional, inelegant, efficient version)

```
M2 = λ t.(if (mt?  t)
        1
        (if (zero?  (num t))
              (EX 0)
              (let ((l (M2 (lson t))))
                    (if (EX? l)
                          (let ((r (M2 (rson t))))
                                (if (EX? r)
                                      r
                                      (* (num t) (* l r))))))))
```

Constructor EX acts like an injection into a sum type NonZeroNum + Zero

Optimized version with control

8.7.3. EXAMPLE. (Non functional, elegant, efficient version)

```
M3 = λ t.  catch j in
       (let L = λ t'.(if (mt?  t')
                      1
                      (if (zero?  (num t'))
                          (throw j 0)
                          (* (num t')
                          (* (L (lson t'))(L (rson t'))))))))
       in (L t))
```

CPS-transformation

- Originally defined by Reynolds 1972 in his study of meta-circular interpreters (e.g. representing CBV in CBN)
- Objectifies the control structure by transforming a function f(x) into f(x,c) where c is a *continuation*
- Has been used to give functional semantics to *control operators* (e.g. Scheme)
- Analogy to the representation of a store by adding an extra argument *globally*

CPS version

8.7.4. EXAMPLE. (CPS version of M3)

```
M4 = λ t.  λ k.
        (if (mt?  t)
              (k 1)
              (if (zero?  (num t))
                    0
                    ((M4 (lson t))
                          (λ l.((M4 (rson t))
                          (λ r.(k (* (num t) (* l r)))))))))
```

Encoding of control in λ_Δ

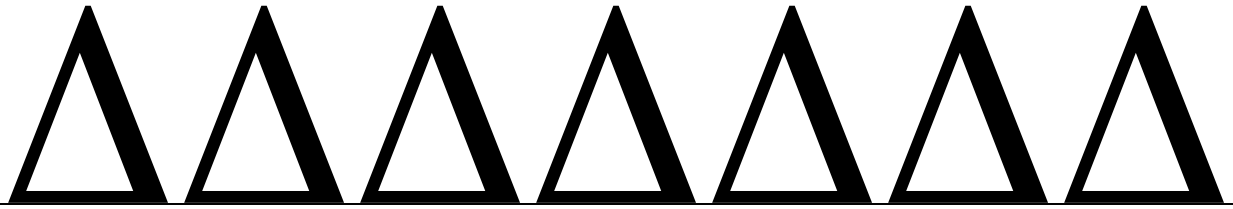
```
M = λt.Δ j.j
    (Y (λf.λt'. (if (mt?  t')
                    1
                    (if (zero?  (num t'))
                        ε(j 0)
                        (* (num t')
                        (* (f (lson t'))(f (rson t'))))))) t)
```

where $\mathbf{Y}$ denotes Church's fixpoint combinator and $\varepsilon(M)$ abbreviates $\Delta x.M$, for $x \notin \mathrm{FV}(M)$.

It is instructive to verify that, e.g., for $\mathtt{T} \equiv \mathtt{<2,<0,nil,nil>,nil>}$, we have $\mathtt{MT} = 0$, noticing how an "exception" is raised as the node value 0 is encountered.

Summary

- *The operator Δ associated with double-negation elimination is a control operator*
- *The proof embedding **t** corresponds to a well-known CPS-transformation that represents control in purely functional terms*
- *The logical embedding **k** of CL into IL is just the accompanying type transformation*



Danke
für die
Aufmerksamkeit!