En kort innføring i ML

Tom Therkildsen *
Institutt for informatikk

7. februar 1990
Modifisert 21. januar 1992 †

1 Typer

Vi nytter begreper til å beskrive samlinger av objekter. En slik samling kaller vi gjerne for en *klasse*, en *kategori* eller en *mengde*. Alternativt kan vi si at begrepet beskriver en bestemt *type* av objekter, f.eks. store objekter, røde objekter eller kalde objekter. Typen til et objekt bestemmer det som karakteriserer samlingen av objekter. Vi sier derfor at *en type er et begrep*.

Vi sier videre at en **datatype** er en mengde dataobjekter med felles egenskaper. Slike egenskaper inkluderer hva slags *verdier* objektene i typen kan anta, og hvilke *operasjoner* som er mulige å utføre på dem. Velkjente datatyper er heltall, boolske og karakterer.

2 Reduksjoner

Et ML-program er enkelt sagt et *uttrykk*. Dette uttrykket blir *evaluert* av ML-kompilatoren som returnerer *verdien* og *typen* til uttrykket. Et enkelt eksempel:

- $3 + 7;$

val $it = 10 : int$

Tegnet '-' er ML-kompilatorens klarsymbol. Den første linjen er et lovlig ML-program mens den neste er den typede *verdien* til programmet. Observer at programmet slutter med et semikolon. ML inneholder alle de velkjente "primitive" datatypene som heltall, boolske, reelle tall og tegnstrenger. Negative tall benevnes (noe uvant) ved å prefikse med tegnet '~' isteden for det mer velkjente '−':

- $2 - 4;$

val $it =$ ~$2 : int$

Talltegn har seg selv som verdi:

- $13;$

val $it = 13 : int$

Uttrykk som har denne egenskapen kaller vi **kanoniske**. Når ML leser et ikke-kanonisk uttrykk, vil den forsøke å *redusere* uttrykket til sin kanoniske form. Dette gir samtidig en forklaring på hva en *beregning* er i ML's forstand. Se på følgende eksempel:

*Denne framstillingen er basert på boka *Functional Programming Using Standard ML* av Åke Wikström. De fleste eksemplene er også hentet derfra.

†Endringene er foretatt av Magne Haveraaen, Institutt for informatikk. Noen feil er rettet, teksten er utvidet med flere av ML's konstruksjoner og den er tilpasset *Standard ML of New Jersey*, som nå er installert under navnet **sml** på FA og HA ved Institutt for informatikk. Denne kompilatoren for ML er utviklet av David MacQueen, AT&T Bell Labs., Murray Hill, NJ, USA

– $2 + 3 + 4$;

val $it = 9 : int$

Vi lar tegnet $\longrightarrow$ bety "reduseres til". I så fall kan beregningen over beskrives ved hjelp av følgende to reduksjonsserier:

$$2 + 3 + 4 \longrightarrow 5 + 4 \longrightarrow 9$$

$$2 + 3 + 4 \longrightarrow 2 + 7 \longrightarrow 9$$

Observer at en reduksjon aldri endrer *verdien* til et uttrykk, den bare *forenkler* uttrykkets form. I ML er en beregning det samme som en serie reduksjoner. Når et gitt uttrykk ikke lenger kan reduseres, er beregningen over. Uttrykket har da antatt sin kanoniske form. Det er mulig å lage uttrykk som fører til *ikke-terminerende* beregninger. Verdien til slike uttrykk er udefinert.

Presedens-relasjoner bestemmer hvilken rekkefølge reduksjonene skal foretas i, f.eks. kan vi både gjøre

$$2 + 3 * 4 \longrightarrow 5 * 4 \longrightarrow 20$$

og

$$2 + 3 * 4 \longrightarrow 2 + 12 \longrightarrow 14$$

men reglene for presedens for operatorer er de samme i ML som i matematikken forøvrig, så det er det *siste* alternativet som faktisk utføres. Vi sier at operatorene $*$, *div* og *mod* binder sterkere (har høyere presedens) enn $+$ og $-$. Ønsker vi en annen evaluerings-orden enn den som er gitt ved presedens-reglene, må vi nytte parantessetting.

3 Variabler, bindinger og omgivelser

Identifikatorer i ML konstrueres på "vanlig måte" ved hjelp av karakterer og heltall. Reglene er omtrent som i Pascal: En identifikator er en streng som har en karakter i første posisjon. Strengen kan ikke være identisk med et **resevert ord** i ML. Tildeling av verdi til en identifikator gjøres ved hjelp av det reserverte ordet **val**:

– **val** $pi = 3.1415927$;

val $pi = 3.1415927 : real$

Vi kaller dette en *verdideklarasjon*. Denne verdideklarasjonen innfører en *binding*, nemlig den at identifikatoren *pi* nå er bundet til *verdien 3.1415927*. Systemet benytter klarsymbolet '>' hver gang det foretar en binding. Identifikatoren kan nå nyttes som en vanlig variabel.

Observer at effekten av å utføre en verdideklarasjon er at identifikatoren bindes til *verdien* til det aktuelle uttrykket og ikke til selve uttrykket. Dette er en viktig distinksjon, noe følgende eksempel viser:

– **val** $radius = 5.0$;

val $radius = 5.0 : real$

– **val** $diameter = 2.0 * radius$;

val $diameter = 10.0 : real$

– $pi * diameter$;

val $it = 31.415927 : real$

– **val** $radius = 20.0$;

val $radius = 20.0 : real$

– $pi * diameter$;

val $it = 31.415927 : real$

Verdien til uttrykket $pi * diameter$ er den samme som tidligere, fordi *diameter* fortsatt er bundet til *verdien* til uttrykket $2.0 * radius$, som er 10.0, og *ikke* til *uttrykket* $2.0 * radius$, som er noe annet. Vi må altså skille mellom uttrykk og deres verdier.

Bindinger har form av *ligninger* som ML bruker til å foreta reduksjoner med. Hver gang et ML finner et navn i et uttrykk, vil dette bli erstattet med navnets verdi når ML leter etter en reduksjon, f.eks. er det reduksjonen $pi \longrightarrow 3.1415927 : real$ som gjøres ved:

– pi;

val $it = 3.1415927 : real$

Vi sier at en samling av bindinger utgjør en *omgivelse*. Mens bindinger er ordnede par av identifikatorer og verdier, er omgivelser *funksjoner* fra identifikatorer til verdier. Vi skjønner dermed at verdien til et uttrykk er bestemt av den omgivelsen det evalueres i. Når vi starter ML-kompilatoren befinner vi oss i en standard omgivelse, der endel standard funksjoner som *abs*, *sin*, *cos* og *sqrt* er kjent. Som vanlig sier vi at de delene av et program der en identifikator er kjent utgjør **gyldighetsområdet** (av og til benevnt *skopet*) til identifikatoren. Vanligvis vil dette gå fra det stedet i programmet der identifikatoren tildeles en verdi og til slutten. Men vi kan også snakke om gyldighetsområdet til *bindinger*. Da vil et gyldighetsområde på en identifikator gå fra en verdideklarasjon og til den neste.

ML har en spesial-identifikator kalt "*it*". Denne identifikatoren vil til enhver tid benevne verdien til det *sist evaluerte uttrykket* i programmet:

– $5 + 5$;

val $it = 10 : int$

– it;

val $it = 10 : int$

– $it + it$;

val $it = 20 : int$

4 Funksjoner

La A og B være to mengder. En *relasjon* $R = A \times B$ er mengden av ordnede par (a, b) slik at $a \in A$ og $b \in B$. Denne relasjonen kalles et *kartesisk produkt* eller et *kryssprodukt*. I parene kaller vi alle a for *førstekoordinater* og alle b for *annenkoordinater*. Vi sier at R går fra A til B. Dersom $A = B$, sier vi at R er en relasjon *over* A (B). Dersom R er slik at det for ethvert unikt førstekoordinat bare finnes *ett* annenkoordinat, kaller vi R en *funksjon*.[1] Slik sett er enhver funksjon f fra en mengde A til en mengde B en delmengde av kryssproduktet $A \times B$. Vi kaller A for funksjonens *domene* og B for funksjonens *rekkevidde*. Dersom $A = B$ sier vi at f er en funksjon *i* A (B).

I et programmeringsspråk vil domenet og rekkevidden til en funksjon alltid være en delmengde av objekter i en datatype. Her får vi visse vansker når vi ønsker å omtale funksjoner der domenet eller rekkevidden er av infinitt kardinalitet. Enhver datatype er jo i praksis endelig, men vi ønsker å betrakte dem som reelle motstykker til de semantiske domenene de er avbildet på. F.eks. ønsker vi å snakke om *heltall (integers)* i et program *som om* de hadde samme egenskaper som heltallene i sin allminnelighet. Dette får vi til ved å se bort fra problemer som er knyttet til over- og underflyt og approksimasjon av f.eks. reelle tall.

Infinitte funksjoner er også vanskelige å *beskrive* i en datamaskin fordi alt som kan beskrives der må være endelig. Her er begrepet om *variable* nyttig. Det er kjent (siden Euler) at uttrykket $n * (n + 1)/2$ benevner summen av alle positive heltall fra 1 og opp til tilfeldig n. På den måten kan vi la uttrykket

$$f(n) = n * (n + 1)/2$$

[1] Observer at dette ikke utelukker at et annetkoordinat kan være parret med mer enn ett førstekoordinat.

stå for den infinitte funksjonen i mengden av positive heltall som vi ønsker å beskrive. I ML kan vi skrive det samme ved hjelp av

– **fun** $euler(x) = x * (x + 1)\ div\ 2;$

val $euler =$ **fn** $: int -> int$

Vi ser at resultatet av **fun**-setningen er at identifikatoren *euler* bindes til en *verdi*, men at denne verdien er en *funksjon* i heltall, noe som går fram av notasjonen **fn** $: int-> int$. Observer at selv om ML er sterkt typet, så vil kompilatoren vanligvis dedusere typene på egen hånd slik at vi bare unntaksvis må deklarere dem selv. Identifikatoren *euler* er altså bundet til en *funksjon*, og denne funksjonen kan igjen *anvendes* på argumenter:

– $euler(3);$

val $it = 6 : int$

Grunnen til at ML returnerer med det reserverte ordet **fn** isteden for f.eks. å gjengi selve *funksjonskroppen*, er at hensikten med språket er å understøtte s.k. *funksjonell abstraksjon.* Tanken her er at mens en *prosedyre* en en oppskrift på hvordan et resultat skal framkomme, er en funksjon en s.k. *black box* der vi ser bort fra på hvilken måte resultatene blir produsert. For eksempel kan jo en funksjon som *euler* beregnes på mange måter. Det interessante er imidlertid at vi får det rette svaret, og ikke måten det framkommer på. ML ordner funksjonell abstraksjon ved å "se bort fra" funksjonskroppen til en funksjon så snart den er definert. Den "gjemmes" bak det reserverte ordet **fn**.

En bedre forståelse av hva en funksjon er får vi kanskje ved å vise til at **fun**-konstruksjoner egentlig er en forkortelse for et **val**-uttrykk. Vi kan nemlig definere *euler* på denne måten også:

– **val** $euler =$ **fn** $x => x * (x + 1)\ div\ 2;$

val $euler =$ **fn** $: int -> int$

Et funksjonsuttrykk som **fn** $(x : int) => 3*x$ benevner en funksjon på samme måte som talltegnet '3' benevner tallet 3. For de som er fortrolige med LISP eller λ-kalkyle, er det verdt å merke seg at et **fn**-uttrykk er en λ-abstraksjon. Funksjonen over kan skrives slik i typet λ-kalkyle:

$$euler = \lambda x : int\ .\ x * (x + 1)\ div\ 2$$

Verdien til et funksjonsuttrykk er en funksjon, og denne kan vi igjen fore med argumenter:

– (**fn**$(x : int) => 3 * x)3;$

val $it = 9 : int$

Her ser vi at det å anvende en funksjon på et argument i ML kan beskrives enkelt ved å la argumentet etterfølge funksjonen. Derfor er bruk av *paranteser* strengt tatt ikke nødvendig slik det er i matematikken. Vi kan like gjerne skrive *euler* 3 som $euler(3)$. Det samme gjelder i definisjonen av funksjoner:

– **val** $double =$ **fn** $x : int => 2 * x;$

val $double =$ **fn** $: int -> int$

– $double\ 2;$

val $it = 4 : int$

Et funksjonsuttrykk er en *operator* i ML (på linje med + og *div*), og denne kan anvendes på en *operand* av passende type. Av alle operatorer i ML er funksjonsuttrykk de som binder sterkest, noe følgende eksempler viser:

– $euler(2 + 3);$

val $it = 15 : int$

$-$ $euler\ 2+3;$

val $it = 6 : int$

I det første tilfellet vil parantesene tvinge ML til å evaluere $(2+3)$ før resultatet gis til funksjonen *euler*. I det andre tilfellet derimot, vil ML legge til 3 til resultatet av uttrykket *euler* 2, hvilket gir $6 : int$ som resultat.

Vi kan forstå funksjonsberegning i ML på to måter: Den enkleste måten er den s.k. *substitusjonsmodellen* der vi sier at argumentene til funksjonen erstatter parametrene i funksjonskroppen:

$$double(10+10) \longrightarrow double(20) \longrightarrow 2*20 \longrightarrow 40$$

Denne modellen er enkel, men er desverre ikke tilstrekkelig til å forklare alle de beregninger vi ønsker utført. Av den grunn finnes det en noe mer komplisert evaluerings-modell som involverer bindinger og omgivelser, og der vi essensielt sier at en funksjonsevaluering skaper en ny omgivelser med spesielle bindinger.

Reglene for en funksjonsanvendelse **F E**, der **F** er en operator og **E** en operand er som følger:

1. Evaluer operatoren **F**. Den verdien som framkommer må være et funksjonsobjekt på formen **fn ID => E'**, der **ID** er et parameter og **E'** er funksjonskroppen.

2. Evaluer operanden **E**. Verdien som framkommer, **V**, er argumentet til funksjonen.

3. Substituer verdien **V** for alle forekomster av parameteret **ID** i funksjonskroppen **E'** og evaluer det resulterende uttrykket. Den verdien som framkommer kaller vi *verdien* til funksjonsanvendelsen.

Et eksempel: Vi definerer en funksjon f, og ser hvordan anvendelsen $f(2+3)$ evalueres ved hjelp av substitusjon.

$-$ **val** $f =$ **fn** $x => 2*x*x;$

val $f =$ **fn** $: int -> int$

Hvilket gir følgende substitusjoner og reduksjoner:

$$f(2+3) \longrightarrow (\mathbf{fn}\ x => 2*x*x)(2+3) \longrightarrow (\mathbf{fn}\ x => 2*x*x)5 \longrightarrow 2*5*5 \longrightarrow 10*5 \longrightarrow 50$$

ML er et sterkt typet språk. Det innebærer at kompilatoren vil insistere på at alle de dataobjekter som skal manipuleres har en kjent type. Det vil også bli foretatt en sjekk på at typene til delene i et uttrykk passer sammen. Dersom ML ikke klarer å avgjøre typen til et uttrykk uten mer informasjon, vil kompilatoren gi beskjed om dette. På samme måte vil kompilatoren protestere dersom det finnes typekonflikter i uttrykk.

Vi kan typebestemme både *domenet* og *rekkevidden* til en funksjon. En definisjon på formen

$-$ **val** $sq =$ **fn** $x : int => x*x;$

val $sq =$ **fn** $: int -> int$

forteller ML at typen til *funksjonen sq* skal være *int*, mens definisjonen

$-$ **val** $sq =$ **fn** $(x : int) => x*x;$

val $sq =$ **fn** $: int -> int$

sier at typen til *argumentet* til *sq* skal være av typen *int*. Vi bestemmer altså typen til funksjonen ved *ikke* å bruke paranteser, og motsatt: Typen til argumentet blir bestemt ved å bruke paranteser. Observer at i begge tilfeller vil ML automatisk slutte typen til den delen av et uttrykk som du ikke eksplisitt har angitt typen til.

ML's rigide typesystem er en fordel ved konstruksjon av store programmer, men de fleste finner forholdet mellom typen *real* og *int* problematisk, fordi vi som regel er interessert i å få en automatisk konvertering, slik at *sq* returnerer en *real* dersom den fores med en *real* og en *int* dersom den fores med en *int*.

5 Kontrollsetninger og bool'ske uttrykk

ML inneholder de vanlige relasjons-operatorene $=, <, >, <=, >=, <>$, som har den vanlige meningen. Typen til et uttrykk som $2 < 3$ er $true : bool$ i ML. Typen *bool* inneholder bare to dataobjekter, nemlig sannhetsverdiene $true$ og $false$. Det finnes ingen *ordning* mellom disse, slik det f.eks. er i Pascal. Derfor er semantikken til elementene i *bool* definert ved:

$$\textbf{val } it = true = true, false = false$$

$$\textbf{val } it = true <> false$$

Et uttrykk som $true >= false$ har ingen mening i ML.

Ved hjelp av typen *bool* kan vi lage kontrollsetninger i ML. For eksempel kan vi definere en funksjon *abs* (innebygget i sml) på følgende måte:

$$-\ \textbf{val } abs = \textbf{fn } x : int => \textbf{if } x >= 0 \textbf{ then } x \textbf{ else } \tilde{}x;$$

$$\textbf{val } abs = \textbf{fn } : int -> int$$

$$-\ abs\ 2;$$

$$\textbf{val } it = 2 : int;$$

$$-\ abs\ \tilde{}2;$$

$$\textbf{val } it = 2 : int$$

For å øke lesbarheten til definisjoner som inneholder kontrollsetninger, er det en god regel å nytte *innrykk*:

$$\begin{array}{l} \textbf{val } abs = \textbf{fn } x : int => \textbf{if } x >= 0 \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{then } x \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \tilde{}x; \end{array}$$

Kontrollsetninger kan også nøstes inn i hverandre:

$$\begin{array}{l} \textbf{val } signum = \textbf{fn } x : int => \textbf{if } x > 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else if } x = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \tilde{}1; \end{array}$$

Når ML evaluerer en kontrollsetning, vil den først evaluere den bool'ske delen av uttrykket. Dersom dette evalueres til *true* vil **then**-delen av uttrykket bli evaluert. I motsatt fall velger ML **else**-delen. Observer at dette innebærer at bare *en* av disse delene blir evaluert.

Ved siden av kontrollsetninger, kan typen *bool* også nyttes til å definere funksjoner som tar argumenter av andre typer og returnerer en sannhetsverdi. Slike funksjoner kaller vi *predikater*. Et eksempel på et predikat kan være en funksjon som tar et heltall som argument og avgjør om tallet er negativt eller ikke:

$$-\ \textbf{val } neg = \textbf{fn } x : int => x < 0;$$

$$\textbf{val } neg = \textbf{fn } : int -> bool$$

I et bool'sk uttrykk er både operatoren og operandene av type *bool*. Velkjente uttrykk av denne typen er *konjunksjoner* ("og") og *disjunksjoner* ("eller"). For eksempel kan vi lage et uttrykk som avgjør om et heltall er større eller lik 0 og mindre enn 10:

$$-\ \textbf{val } digit = \textbf{fn } (x : int) => 0 <= x \textbf{ andalso } x < 10;$$

$$\textbf{val } digit = \textbf{fn } : int -> int$$

For at denne funksjonen skal avgi verdien *true* må *begge* vilkårene i uttrykket være tilfredsstilt. Vi har benyttet en konjunksjon. Disjunksjoner lages tilsvarende ved:

– **val** $newdigit$ = **fn** $(x : int)$ => $0 > x$ **orelse** $x >= 10$;

val $newdigit$ = **fn** : $int -> int$

Her er det tilstrekkelig at det aktuelle argumentet *enten* er mindre enn 0 eller større eller lik 10. Standardfunksjonen *not* reverserer sannhetsverdien til ethvert bool'sk uttrykk:

– **val** $negate$ = **fn** $(x : bool)$ => $not(x)$;

val $negate$ = **fn** : $bool -> bool$

– $negate\ true$;

val $it = false : bool$

Gitt definisjonen av *newdigit* kunne vi fått til *digit* ved å gi:

– **val** $digit$ = **fn** $(x : int)$ => $not(newdigit\ x)$;

Her ser vi at *digit* blir evaluert ved å reversere sannhetsverdien til *newdigit*. Det betyr at alle tall som faller inn under *newdigit* ikke faller inn under *digit* og vice versa. Observer også at vi kan definere en funksjon ved å gjøre bruk av en *annen*. Dette er nyttig for å få til abstraksjoner.

6 Par og tupler

En enkel datastruktur i ML er *par*. Typen til et par som $(3, true)$ benevnes med $int * bool$. Observer at tegnet $*$ i denne sammenhengen er en *type konstruktør* og ikke multiplikasjonsoperatoren. Konstruktøren for par benevnes med komma og paranteser. Dersom det finnes n_1 objekter av en type T_1 og n_2 objekter av en type T_2, så finnes det $n_1 * n_2$ objekter av typen $T_1 * T_2$:

– $("tom", 1)$;

val $it = ("tom", 1) : string * int$

Et objekt av typen par er altså et element i kryssproduktet mellom de to typene som paret er konstruert fra. Komponentene i et par kan være av en hvilken som helst type, f.eks. av typen par. Det betyr at vi kan lage par av par:

– $("tom", (1.5, true))$;

val $it = ("tom", (1.5, true)) : string * (real * bool)$

Også funksjoner kan være komponenter i par:

– **val** sqr = **fn** $(x : real)$ => $x * x$;

val sqr = **fn** : $real -> real$

– $(sqr, true)$;

val it = (**fn**, $true$) : $(real -> real) * bool$

Vi kan også navngi par:

– **val** $origo = (0.0, 0.0)$;

val $origo = (0.0, 0.0) : real * real$

– **val** $person = ("tom", 29)$;

val $person = ("tom", 29) : string * int$

– $(origo, person)$;

val $it = ((0.0, 0.0), ("tom", 29)) : (real * real) * (string * int)$

Observer at ML vil evaluere verdien til komponentene i et par, på samme måte som den evaluerer ethvert annet uttrykk.

ML er et språk der objekter av enhver type er s.k. *førsteklasses borgere*. En annen måte å si dette på, er at alle typer i ML er *komplette*. Det betyr bl.a. at ethvert dataobjekt kan være

- argument til en funksjon
- resultatet av en funksjonsanvendelse
- navngitt i en definisjon
- en del av en større datastruktur som par, lister eller records

Når vi skal ha tilgang til komponentene i en datastruktur, må vi nytte spesielle *selektorer* som er definert for den typen struktur vi arbeider med. Anta for eksempel at vi ønsker å lage en funksjon *max* som identifiserer det største heltallet i ethvert par av heltall. Da må vi ha tilgang til komponentene i slike par, og dette kan gjøre på *to* måter. Den enkleste er ved bruk av *mønster-gjenkjenning*. Et *mønster* som (x, y) kan legges "opp på" et par med den effekten at den første komponenten kan refereres ved x og den andre ved y:

$- \textbf{val}\ max = \textbf{fn}\ (x, y) : int * int => \ \textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y;$

$\textbf{val}\ max = \textbf{fn}\ \ : (int * int)\ -> int$

$-\ max(2, 100);$

$\textbf{val}\ it = 100 : int$

Den andre måten er ved eksplisitt introduksjon av selektorene **fst** og **snd** som gir tilgang til henholdsvis første og andre komponent (*max* er innebygget i sml):

$- \textbf{val}\ fst = \textbf{fn}\ (x, y) => x;$

$\textbf{val}\ fst = \textbf{fn} : ('a * 'b)\ -> 'a$

$- \textbf{val}\ snd = \textbf{fn}\ (x, y) => y;$

$\textbf{val}\ snd = \textbf{fn} : ('a * 'b)\ -> 'b$

$- \textbf{val}\ max = \textbf{fn}\ p : int * int => \textbf{if}\ fst\ p > snd\ p\ \textbf{then}\ fst\ p\ \textbf{else}\ snd\ p;$

$\textbf{val}\ max = \textbf{fn}\ \ : (int * int)\ -> int$

Dersom vi studerer ML's respons på definisjonene av **fst** og **snd**, ser vi at *typen* til disse funksjonene vil variere med hva slags argumenter de gis. Dette kaller vi gjerne for *type polymorfisme* og ML signaliserer at typen til en funksjon er polymorf ved å returnere signaturer på formen $'a * 'b -> 'a$. Navn av typen $'a$ er det vi kaller *typevariabler*, m.a.o. variabler som kan anta enhver kjent type. Vi kjenner igjen slike variabler ved at de alltid har tegnet "'" først i navnet. Funksjoner som tilordnes uttrykk som inneholder en eller flere typevariabler sier vi er polymorfe, hvilket betyr at de kan anta flere ulike typer. Dette er også en nyttig abstraksjonsmekanisme.

Alle par kan testes for likhet og ulikhet ved hjelp av operatorene = og <>. Imidlertid finnes det ingen predefinert ordning mellom par av samme type. Ønsker vi å ha en slik ordning, f.eks. for par av heltall, må vi programmere operatorer som gir det rette resultatet selv:

$$\begin{aligned} \textbf{val}\ lt = \textbf{fn}\ ((x1 : int, x2 : int), (y1, y2)) => {} & x1 < y1\ \textbf{orelse} \\ & x1 = y1\ \textbf{andalso}\ x2 < y2; \end{aligned}$$

Observer at typen til dette predikatet er $((int * int) * (int * int))\ -> bool$, og at ML *selv* deduserer typen til det andre paret ut fra den informasjonen som er gitt om det første.

Begrepet par kan utvides til ikke bare å omfatte kryssprodukt av 2 typer, men til kryssproduktet av n typer for vilkårlige positive n. Et slikt uttrykk kalles et ***n*-tuppel**.

$- ("tom", 29, origo, 4 <= 2);$

$\textbf{val}\ it = ("tom", 29, (0.0, 0.0), false) : string * int * (real * real) * bool$

Her har vi et 4-tuppel der tredje komponent er et par (eller 2-tuppel).

7 Semantikk

En enkel *operasjonell* semantikk for ML får vi ved å anta at vi disponerer en abstrakt reduksjonsmaskin som er istand til å foreta reduksjoner på velformede uttrykk skrevet i ML. Denne maskinen består av to komponenter: Et *programlager* der det programmet som skal utføres ligger lagret, og en *kontrollenhet* som inneholder mengden av lovlige reduksjoner som kan utføres på velformede ML-program.

Vi angir semantikken til uttrykk i ML ved å beskrive hvordan reduksjonsmaskinen vil behandle dem. Enkelt sagt vil maskinen foreta reduksjoner på et gitt uttrykk helt til det har antatt sin kanoniske form. Denne formen er en beskrivelse av uttrykkets *verdi*, men som regel sier vi bare at den kanoniske formen *er* verdien til uttrykket. [2] Kanoniske uttrykk kan ikke reduseres ytterligere. De har seg selv som "verdi".

Vi ser først på hvordan maskinen behandler uttrykk som inneholder paranteser:

$$4 * (2 + 3) \longrightarrow 4 * (5) \longrightarrow 4 * 5 \longrightarrow 20$$

Vi ser at den andre reduksjonen essensielt består i å fjerne parantesene. Legg merke til at dette skrittet *ikke* er mulig i den første reduksjonen fordi vi da vil endre meningen til uttrykket. I den andre reduksjonen kan parantesene fjernes fordi de da omslutter et kanonisk uttrykk. Vi setter opp følgende regler:

1. Dersom e er et uttrykk og $e \longrightarrow e\prime$ er en lovlig reduksjon, så er også $(e) \longrightarrow (e\prime)$ en lovlig reduksjon.

2. Dersom v er et kanonisk uttrykk, så er $(v) \longrightarrow v$ en lovlig reduksjon.

I ML finnes en rekke binære aritmetiske operatorer, som $+, *, div, mod$ og $-$. Vi benytter tegnet $\odot$ som en forkortelse for alle. Reduksjonen av uttrykk som involverer en av disse operatorene kan da gis:

1. Dersom e_1 og e_2 er uttrykk, og $e_1 \longrightarrow e_1\prime$ er en lovlig reduksjon, så er $e_1 \odot e_2 \longrightarrow e_1\prime \odot e_2$ en lovlig reduksjon.

2. Dersom n er et talltegn og $e_2 \longrightarrow e_2\prime$ en lovlig reduksjon, så er $n \odot e_2 \longrightarrow n \odot e_2\prime$ en lovlig reduksjon.

3. Dersom n_1 og n_2 er talltegn, så er $n_1 \odot n_2 \longrightarrow n_3$, en lovlig reduksjon, der n_3 er *resultatet* av å utføre $n_1 \odot n_2$ for $\odot = +, *, -, div, mod, \ldots$.

Å evaluere *kontrollsetninger* forutsetter at alle bool'ske uttrykk kan reduseres til sin kanoniske form, som er enten *true* eller $false$. Disse reduksjonene er gitt ved induksjon over sannhetsfunksjonene til **andalso**, **orelse** og **not**. ML vil *først* redusere den bool'ske delen av en kontrollsetning ved hjelp av disse reglene, og deretter evaluere en av forgreningene avhengig av om den første reduksjonen resulterte i *true* eller $false$. [3] Vi lar variablene b og $b\prime$ ha bool'ske uttrykk som rekkevidde, mens variablene e_1 og e_2 går over generelle uttrykk:

1. Hvis $b \longrightarrow b\prime$ er en lovlig reduksjon, så er **if** b **then** e_1 **else** $e_2 \longrightarrow$ **if** $b\prime$ **then** e_1 **else** e_2 også en lovlig reduksjon.

2. **if** *true* **then** e_1 **else** $e_2 \longrightarrow e_1$ er en lovlig reduksjon.

3. **if** $false$ **then** e_1 **else** $e_2 \longrightarrow e_2$ er en lovlig reduksjon.

[2] Hvilket forsåvidt er riktig, sett fra reduksjonsmaskinens ståsted.

[3] Dette er ett av ytterst få tilfeller der ML har en *bedagelig* (lazy) adferd. I regelen vil ML insistere på å evaluere komplette uttrykk, men her sparer en vesentlig eksekveringstid ved bare å forfølge *en* av forgreningene i en **if-then-else** sats. Dette er dessuten helt nødvendig om uttrykk som **if** $x <> 0$ **then**x div 0 **else** 0 skal ha noen mening. De bool'ske operatorene **andalso** og **orelse** evalueres også bedagelig fra venstre mot høyre.

Vi kan se hvordan ML evaluerer kontrollsetninger ved hjelp av reduksjoner gjennom et eksempel:

$$\textbf{if } 2 < 4 \textbf{ then } 2+3 \textbf{ else } 1 \textit{ div } 0 \longrightarrow$$

$$\textbf{if } \textit{true} \textbf{ then } 2+3 \textbf{ else } 1 \textit{ div } 0 \longrightarrow 2+3 \longrightarrow 5$$

Observer at det semantisk ukorrekte uttrykket 1 *div* 0 aldri blir forsøkt evaluert.

For å forstå meningen til uttrykk som involverer *variabler*, må vi bygge ut reduksjonsmaskinen med et ekstra lager, *omgivelseslageret*. I dette lageret finnes alle *bindinger* som er innenfor gyldighetsområdet til det aktuelle uttrykket. Dersom ML skal redusere et uttrykk som $x + 3$, vil kompilatoren først lete i lageret for å se hvilken verdi variabelen x er bundet til. Hver gang vi foretar en ny verdideklarasjon blir omgivelseslageret bygget ut. Legg merke til at vi også kan *endre* bindingene i en omgivelse under kjøretid. I så fall *slettes* den gamle bindingen og den nye innføres.

Vi skal la notasjonen $id = v$ bety at identifikatoren id er bundet til verdien v. En omgivelse vil vanligvis bestå av flere bindinger, og det benevner vi med

$$\{id_1 = v_1, id_2 = v_2, \ldots, id_n = v_n\}$$

En omgivelse er altså en *mengde* av bindinger, der hver id_i bare kan forekomme en gang. Vi *overskriver* omgivelser ved å legge til nye bindinger. Dersom Env_1 og Env_2 er omgivelser der følgende gjelder:

$$Env_1 = \{x = 3, y = 8\}$$

$$Env_2 = \{y = 5\}$$

så skal notasjonen $Env_1 \oplus Env_2$ bety at alle bindinger i Env_2 som angår identifikatorer som også finnes i Env_1 overskriver disse. Det vil si:

$$\{x = 3, y = 8\} \oplus \{y = 5\} \longrightarrow \{x = 3, y = 5\}$$

Den verdien som er assosiert med identifikatoren id i omgivelsen Env, benevner vi med $Env(id)$. Ut fra dette skjønner vi at omgivelser egentlig ikke er annet enn *funksjoner* fra identifikatorer til verdier.

Vi skriver $\langle e \mid Env \rangle$ for å evaluere et uttrykk e i en omgivelse Env. Vi kan nå si hva det betyr å evaluere en identifikator i en omgivelse:

1. Hvis $Env(id) = v$, så er $\langle id \mid Env \rangle \longrightarrow \langle v \mid Env \rangle$ en lovlig reduksjon. Altså: Dersom id er bundet til v i en omgivelse Env, så er v *verdien* til uttrykket id i denne omgivelsen.

Ettersom ethvert uttrykk i ML kan inneholde variabler, må vi nå utvide samtlige regler til å omfatte omgivelser. Vi skal ikke gi detaljene for dette her.

"Verdien" til en verdideklarasjon er en omgivelse som inneholder de bindingene som deklarasjonen innfører. Dersom Env er den omgivelsen som eksisterte *før* deklarasjonen ble utført, kan vi gi følgende regler for evaluering av verdideklarasjoner:

1. Hvis $\langle e \mid Env \rangle \longrightarrow \langle e\prime \mid Env \rangle$ er en lovlig reduksjon, så er også $\langle \textbf{val } id = e \mid Env \rangle \longrightarrow \langle \textbf{val } id = e\prime \mid Env \rangle$ en lovlig reduksjon.

2. Når uttrykket e er evaluert, kan vi lage den resulterende omgivelsen. Det vil si at $\langle \textbf{val } id = v \mid Env \rangle \longrightarrow \langle \{id = v\} \mid Env \rangle$ er en lovlig reduksjon.

Når en deklarasjon er utført, må den resulterende omgivelsen *kombineres* med den omgivelsen som eksisterte tidligere. Det er den kombinerte omgivelsen som er omgivelsen for den *neste* deklarasjonen, og så videre. Reduksjonsmaskinen vil altså "akkumulere" omgivelser.

La oss endelig se hvordan vi kan gi en semantikk for funksjoner. Det innebærer at vi både må forklare hvordan funksjoner *defineres* og hvordan de *anvendes* på argumenter. Dersom vi ser på en funksjonsdeklarasjon på formen

– **val** f = **fn** $x => x + 3$;

val f = **fn** : $int -> int$

så går det tydelig fram at en funksjon er et objekt i seg selv. En funksjonsdeklarasjon har altså samme mening som en verdideklarasjon, den etablerer en binding mellom et funksjonsnavn og en funksjonskropp. Neste skritt er å se på *anvendelsen* av en funksjon på et argument, f.eks. anvendelsen f 10. For å få til denne anvendelsen må både operatoren f og operanden 10 evalueres. Uttrykk evalueres fra høyre mot venstre:

1. Hvis $\langle e_1 \mid Env\rangle \longrightarrow \langle e_1\prime \mid Env\rangle$ er en lovlig reduksjon, så er også $\langle e_1\ e_2 \mid Env\rangle \longrightarrow \langle e_1\prime\ e_2 \mid Env\rangle$ en lovlig reduksjon.
2. Hvis $\langle e_2 \mid Env\rangle \longrightarrow \langle e_2\prime \mid Env\rangle$ er en lovlig reduksjon, så er også $\langle(\textbf{fn}\ id = e)e_2 \mid Env\rangle \longrightarrow \langle(\textbf{fn}\ id = e)e_2\prime \mid Env\rangle$ en lovlig reduksjon. Vi kan altså kombinere uttrykk slik at de former funksjonsanvendelser.

Vi har nå sett hvordan maskinen vil evaluere de individuelle delene av en funksjonsanvendelse. Neste skritt er å se på selve anvendelsen. For å gi en korrekt tolkning av dette, må vi imidlertid innføre tre nye begrep:

fri variabel: Dersom en funksjonsdefinisjon inneholder et *navn* som ikke er et parameter, sier vi at dette navnet (variabelen) er **fri** i funksjonen.

bundet variabel: En variabel i en funksjonsdefinisjon som ikke er fri, sier vi er **bundet**.

lukking: Når vi assosierer bindinger som var gyldige ved *definisjonen* av en funksjon med de fri variable i funksjonsdefinisjonen, sier vi at vi **lukker** funksjonen. Funksjonsdefinisjoner som ikke kan lukkes gir typefeil i ML.

Følgende enkle eksempel illustrerer poenget. Anta at identifikatorene x, y og f ikke er tilordnet noen verdi i den aktuelle omgivelsen:

– **val** f = **fn** $x => x + y$;

*std_in:*2.19 *Error: unbound variable y*

– **val** $y = 3$;

val $y = 3 : int$

– **val** f = **fn** $x => x + y$;

val f = **fn** : $int -> int$

– f 2;

val $it = 5 : int$

– **val** $y = 2$;

val $y = 2 : int$

– f 2;

val $it = 5 : int$

I første linje i dette eksempelet får vi typefeil fordi ML ikke klarer å lukke definisjonen. Den fri variable y er ikke bundet til noen verdi i omgivelsen. Ved å tilordne en verdi til y får vi definisjonen igjennom. Observer imidlertid at *etter* at vi har modifisert denne bindingen, vil anvendelsen f 2 fortsatt gi $5 : int$ som svar, fordi anvendelsen gjøres i henhold til de bindingene som var gyldige da funksjonen ble *definert*. Dette kaller vi *statisk binding*, til forskjell fra *dynamisk binding* slik vi f.eks. finner det i LISP. Reglene for funksjonsanvendelse må derfor inneholde en beskrivelse av hvordan funksjonsdefinisjoner lukkes i den aktuelle omgivelsen:

3. $\langle \textbf{val}\ f = \textbf{fn}\ id => e \mid Env\rangle \longrightarrow \langle\{f = \langle\textbf{fn}\ id => e \mid Env\rangle\} \mid Env\rangle$ er en lovlig reduksjon.

4. Når en anvendelse utføres, må maskinen hente fram igjen den gamle omgivelsen fra den lukkingen som ble gjort da deklarasjonen ble utført: $\langle\langle \mathbf{fn}\ id => e \mid Env\rangle\ v \mid Env\prime\rangle \longrightarrow \langle\langle e \mid Env \oplus \{id = v\}\rangle \mid Env\prime\rangle$ er en lovlig reduksjon.

5. Endelig blir funksjonskroppen evaluert i den nye omgivelsen: Dersom $\langle e \mid Env\rangle \longrightarrow \langle e\prime \mid Env\rangle$ er en lovlig reduksjon, så er $\langle\langle e \mid Env\rangle \mid Env\prime\rangle \longrightarrow \langle\langle e\prime \mid Env\rangle \mid Env\prime\rangle$ også en lovlig reduksjon.

6. Når funksjonskroppen er redusert til et kanonisk uttrykk, blir den temporære omgivelsen oppgitt: $\langle\langle e \mid Env\rangle \mid Env\prime\rangle \longrightarrow \langle v \mid Env\prime\rangle$.

Disse reglene for evaluering blir noe modifisert når vi innfører rekursive funksjoner, men det kommer vi tilbake til.

8 Lokale bindinger

Hovedregelen for gyldighetsområdet til en identifikator i ML er at dersom den er deklarert på topp-nivå, vil den være kjent inntil neste deklarasjon som involverer samme identifikator eller til programmet har stoppet. Dette gir imidlertid få abstraksjonsmuligheter, og derfor inneholder ML også muligheter til å deklarere navn som er *lokale* til uttrykk eller til andre deklarasjoner.

Som et eksempel på det første, kan vi se på formelen som beregner arealet i en trekant med sidene a, b og c:

$$\sqrt{p * (p - a) * (p - b) * (p - c)}$$

der p er definert ved:

$$p = \frac{a + b + c}{2}$$

Den mest effektive beregningen av denne formelen får vi opplagt ved *først* å beregne p, og deretter trekke roten av produktet. For å få til dette, kan vi i ML ha p som en *lokal* identifikator til den funksjonen som skal beregne arealet:

$$\begin{array}{l}\mathbf{val}\ area = \mathbf{fn}\ (a, b, c) => \mathbf{let}\ \mathbf{val}\ p = (a + b + c)/2.0 \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ sqrt(p * (p - a) * (p - b) * (p - c)) \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{end};\end{array}$$

Vi ser at deklarasjonen av *area* inneholder en lokal deklarasjon, nemlig av p. Den lokale deklarasjonen har sitt gyldighetsområde mellom det reserverte ordet **in** og det reserverte ordet **end**. Vi kan ha *mange* deklarasjoner etter en **let**:

$$\begin{array}{l}\mathbf{let}\ \mathbf{val}\ x = 15; \\ \quad\ \ \mathbf{val}\ y = 3; \\ \quad\ \ \mathbf{val}\ q = 8 \\ \mathbf{in}\ x * y\ div\ q \\ \mathbf{end}; \\ 5 : int\end{array}$$

Vi kan også ha lokale deklarasjoner i andre deklarasjoner. Dette er særlig nyttig der hvor vi ønsker å gjemme funksjoner inne i andre funksjoner, slik at de er usynlige fra utsiden. På den måten kan vi unngå mye forvirring med navngivning og modifikasjon av programmer som flere arbeider med samtidig. Som et enkelt eksempel på dette skal vi anta at vi ønsker å skrive en funksjon som for et gitt heltall bestemmer om det er et "jubileumsår". Vi sier at dersom tallet er delelig med 10 eller 25 så kan vi feire jubileum. Til dette trenger vi en lokal funksjon som avgjør om tallet har den ønskede egenskapen:

$$\begin{array}{l}\mathbf{local}\ \mathbf{val}\ divides = \mathbf{fn}\ (x, y) => y\ mod\ x = 0 \\ \mathbf{in}\ \mathbf{val}\ jub = \mathbf{fn}\ year => divides(10, year)\ \mathbf{orelse}\ divides(25, year) \\ \mathbf{end}; \\ \mathbf{val}\ jub : int\ {->}\ bool\end{array}$$

Gyldighetsområdet for denne typen lokale deklarasjoner er at det som deklareres mellom **local** og **in** er kjent i den deklarasjonen som befinner seg mellom **in** og **end**. Observer at ML bare kvitterer med typen til den "ytterste" definisjonen.

I ML kan vi også innføre nye datatyper. Et enkelt eksempel illustrerer introduksjonen av en oppramstype DAY:

– **datatype** $DAY = Mon \mid Tue \mid Wed \mid Thu \mid Fri \mid Sat \mid Sun$;

con $Mon : DAY$

con $Tue : DAY$

...

con $Sun : DAY$

Her har vi innført åtte nye identifikatorer, hvorav *ett* typenavn (DAY) og syv nye typekonstruktører. Hver ny konstruktør er identifisert ved hjelp av det reserverte ordet **con**[4]. Typen er nå kjent for systemet, og vi kan definere funksjoner som gjør bruk av den:

– **val** $workday =$ **fn** $day => day <> Sat$ **andalso** $day <> Sun$;

val $workday =$ **fn** $-> bool$

Fordi DAY er en oppramstype, kunne vi også ha definert funksjonen *workday* ved hjelp av *case-oppsplitting*. Notasjonen for dette i ML er som følger:

$$\begin{array}{lll} \textbf{val}\ work = \textbf{fn}\ d : DAY => & \textbf{case}\ d\ \textbf{of} & Sat => false \\ & \mid & Sun => false \\ & \mid & \_ => true; \end{array}$$

Legg merke til at det siste alternativet i denne **case**-konstruksjonen er voktet med uttrykket "_", som matcher med *ethvert* element i typen[5]. Faktisk kunne vi også ha valgt et *tredje* alternativ, fordi konstruktørene gir seg selv som resultat:

$$\begin{array}{ll} \textbf{val}\ work = \textbf{fn} & Sat => false \\ \mid & Sun => false \\ \mid & \_ => true; \end{array}$$

Alle tre funksjoner er fra typen DAY og til $bool$.

9 Rekursjon over lister

Lister er en viktig datastruktur. I motsetning til *par* og *tupler*, kan lister inneholde et ubestemt antall elementer. Vi konstruerer lister ved hjelp av operatoren *cons* som benevnes med '::' i ML. Denne operatoren tar et *element* og en *liste* som argumenter. En liste kan også benevnes ved hjelp av parantesene [og], og komma mellom elementene:

– $[2, 3]$;

val $it = [2, 3] : int\ list$

– $4 :: [2, 3]$;

val $it = [4, 2, 3] : int\ list$

– **val** $lst = [sin, cos]$;

val $lst = [$**fn**, **fn**$] : (real -> real)\ list$

[4]Observer at disse konstruktørene konstruerer *seg selv* i en viss forstand.

[5]Et slikt tegn kalles gjerne et *wildcard*, - i UNIX er det tilsvarende tegnet "*".

Observer at ML kvitterer med *både* typen til elementene og typen *list*. Ettersom alle typer i ML er førsteklasses objekter, kan vi ha lister av en hvilken som helst type, så lenge det er *samme* type i samme liste. Vi kan ha lister av lister, lister av funksjoner, heltall og hva det måtte være. En liste kan også være *tom*:

– [ ];

val $it = [\,]\ :\ {'a\ list}$

– nil;

val $it = [\,]\ :\ {'a\ list}$

Her ser vi at ML ikke klarer å dedusere typen til elementene i den tomme listen, og derfor gis den en **polytype**, noe vi ser av notasjonen $'a\ list$. Vi kan velge om vi vil kalle den tomme listen *nil* (som er vanlig i f.eks. LISP) eller [ ], som er det navnet ML selv alltid nytter. Når vi lager lister, bygger vi dem opp ved hjelp av *nil* og '::', ettersom enhver liste kan sees på som en utvidelse av den tomme listen. Notasjonen ved hjelp av paranteser og komma er bare en *forkortelse* for dette:

$$[1,2,3,4] \quad \stackrel{\text{def}}{=} \quad 1 :: 2 :: 3 :: 4 :: nil$$

For å kunne analysere innholdet i lister, trenger vi et sett av *selektorer*, og (minst) et predikat som forteller om en liste er tom eller ikke. Dette siste predikatet kaller vi gjerne *null*, og det defineres ved:

val $null$ = **fn** $nil => true$
$\quad\quad\quad\quad |\ \ \_\ \ => false;$

Til å hente ut elementer fra en liste benytter vi de innebyggede funksjonene *hd* (head) og *tl* (tail) som velger ut h.h.v. *første* element i en liste og resten av listen når første element er fjernet. Vi kunne ha definert dem som:

– $hd[1,2,3]$;

val $it = 1 : int$

– $tl[1,2,3]$;

val $it = [2,3]\ :\ int\ list$

Funksjonene *hd* og *tl* er **polymorfe**, de arbeider på lister av enhver type. Dessuten er de bare *partielt definert*, fordi de er udefinerte for den tomme listen.

Vi setter sammen to lister ved hjelp av konkatenerings-operatoren "@" (denne operatoren kalles gjerne *append*). Listene må være av samme type:

– $[1,2,3]$ @ $[4,5,6]$;

val $it = [1,2,3,4,5,6]\ :\ int\ list$

@ og :: er begge venstre-assosiative, og @ har *nil* som identitetselement.

Til slutt nevner vi noen nyttige funksjoner som er basert på de vi har nevnt:

- *rev* (innebygget) reverserer elementene i en liste
- *last* henter fram siste element i en liste
- *len* angir antall elementer i en liste
- *sum* summerer elementene i en liste (dersom typen til elementene tillater det)
- *prod* multipliserer elementene i en liste (med samme restriksjon som over)

Funksjonene i denne listen har en ting felles, de kan lett defineres ved *rekursjon* over lister. Rekursjon er en generell kontrollmekanisme som er basert på prinsippet om *matematisk induksjon*. Det vil si at vi foretar en oppsplitting av funksjonsberegningen i *to* tilfeller: Basistilfellet, som angir funksjonsverdien for et kjent tilfelle, og induksjonsskrittet som angir hvordan funksjonen antar verdi idet den beveger seg *mot* det kjente basistilfellet[6].

Et enkelt eksempel illustrerer poenget: For å beregne *lengden* til en tilfeldig liste kan vi nytte de to tilfellene:

1. Den *tomme listen* har lengde 0.
2. Alle lister som ikke er tomme har lengden 1+ lengden til den listen som er nøyaktig ett element kortere.

Disse enkle reglene kan vi legge inn i en funksjonsdeklarasjon. Observer at den siste regelen innebærer at vi bare kan beregne et *delresultat*: Vi skal legge 1 til lengden av noe vi ikke vet hvor langt er! Denne delen av funksjonsdeklarasjonen kalles rekursjonskallet:

$$\begin{array}{l} \textbf{val rec } \mathit{len} = \textbf{fn } \mathit{lst} \ : \mathit{'a\ list} => \textbf{if } (\mathit{null\ lst}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } 1 + \mathit{len}(\mathit{tl\ lst}); \end{array}$$

Evaluering av en slik funksjon gjøres ved hjelp av en *stakk*. Vi ser at funksjonen kaller seg selv, men at argumentet for hvert kall er ett element kortere på grunn av funksjonen *tl*. Det reserverte ordet **rec** tillater oss å benevne funksjonsnavnet inne i definisjonen av den samme funksjonen[7]. Observer at selv om *tl* er udefinert for den tomme listen (som etterhvert blir resultatet), så blir dette tilfellet håndtert av **if**-delen av funksjonen. Hver gang listen gjøres ett element kortere, øker dens resultat med 1, hvilket et akkurat hva vi ville. Når basetilfellet *nil* oppstår, vil funksjonen legge 0 til alle de 1'ere som er akkumulert i løpet av prosessen.

Vi ser at deklarasjonen nytter den polymorfe typevariabelen *'a* for å angi at funksjonen skal være definert for lister med enhver type av elementer. Denne informasjonen er imidlertid ML istand til å finne fram til *selv* fordi *nil* og :: også er polymorfe. En kortform av deklarasjonen over er derfor denne:

$$\begin{array}{l} \textbf{val rec } \mathit{len} = \textbf{fn } \mathit{nil} => 0 \\ \qquad\qquad\qquad\quad \mid (x :: xs) => 1 + \mathit{len}\ xs; \end{array}$$

I denne versjonen sjekker *len* direkte på *nil* først, og dersom denne grenen ikke slår til, vet vi at den aktuelle listen *minst* består av *ett* element, etterfulgt av en liste (som kan være *nil*). Notasjonen $(x :: xs)$ er et *mønster* som vi legger på listen, der x refererer til første element og xs til resten. Vi kan dermed be *len* legge til 1 og arbeide videre på den listen som xs refererer til!

Samme type tenkning er naturlig for *rev* som reverserer elementene i en liste:

1. Å reversere en tom liste gir en tom liste.
2. Å reversere en ikke-tom liste er det samme som å legge første element i listen *bak* resultatet av å reversere de gjenværende elementene. Dette får vi til ved hjelp av @.

Disse reglene gir grunnlag for følgende deklarasjon:

$$\begin{array}{l} \textbf{val rec } \mathit{rev} = \textbf{fn } \mathit{nil} => \mathit{nil} \\ \qquad\qquad\qquad\quad \mid (x :: xs) => \mathit{rev}\ xs\ @\ [x]; \end{array}$$

[6]Observer at det siste kravet er helt avgjørende. Dersom funksjonen ikke beveger seg mot det kjente tilfellet, vil den aldri kunne terminere

[7]Dette ordet er innført fordi prinsippet for **val**-setninger brytes ved deklarering av rekursive funksjoner. Generelt er det som deklareres i en **val**-setning bare synlig *etter* at setningen er evaluert. Men i dette tilfelle ville jo evaleringen aldri lykkes fordi funksjonsnavnet refereres inne i den samme deklarasjonen. ML må derfor få beskjed om å behandle dette på en spesiell måte.

Denne funksjonen er av type ($'a\ list$) $->$ ($'a\ list$). La oss derfor som et siste eksempel i denne omgang se på funksjonen *sum* som tar en liste av heltall og returnerer summen av dem. Som vanlig setter vi opp basetilfellet og induksjonsskrittet først.

1. Summen av en tom liste er 0.
2. Summen av en ikke-tom liste er lik verdien til det første elementet pluss summen av resten av listen.

Da kan vi lage denne deklarasjonen:

$$\begin{array}{l} \textbf{val rec}\ sum = \textbf{fn}\ nil => 0 \\ \qquad\qquad\qquad\quad \mid (x :: xs) => x + sum\ xs; \end{array}$$

Hvis vi deklarer funksjoner ved hjelp av **fun** notasjonen, blir det automatisk antatt at funksjonen kan være rekursiv. Vi kan derfor definere *sum* funksjonen som

$$- \textbf{ fun}\ sum\ x = \textbf{case}\ x\ \textbf{of}\ nil => 0 \mid (y :: ys) => y + sum\ ys;$$

$$\textbf{val}\ sum = fn : int\ list\ -> int$$

uten å bruke nøkkelordet **rec**.

Vi så tidligere at semantikken for en funksjonsanvendelse gikk ut på at funksjonen ble evaluert i den omgivelsen som var gyldig *før* funksjonen ble deklarert. Dette fikk vi til ved å foreta en *lukking* av funksjonen, slik at vi fikk fastlagt verdier for fri variable. Når vi definerer rekursive funksjoner, må navnet til funksjonene være kjent inne i selve deklarasjonen. Derfor må lukkingen av rekursive deklarasjoner inkludere en binding til funksjonsnavnet selv. Dette gir denne typen deklarasjoner sitt selvrefererende preg, fordi funksjonsnavnet bindes til den lukkingen som deklarasjonen selv er en del av. Effekten av dette er svært virkningsfull: Vi kan benevne et funksjonsnavn inne i definisjonen av den samme funksjonen!

10 Generell rekursjon og høyere-ordens funksjoner

Operatoren @ tar to lister og konkatenerer dem. La oss nå definere en funksjon *append* som gjør det samme med et *par* av lister. Vi må da modifisere tilnærmingen til rekursjon noe, fordi argumentet til denne funksjonen er et par og ikke en enkelt liste. Det vi ønsker er å kunne gå fra

$$([a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_m])$$

og til

$$([a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m])$$

Når vi bygger en liste, legger vi ett og ett element inn i fronten av listen. Vi kan derfor få til funksjonen vår ved å bryte ned den første listen i sine enkelte elementer, og deretter legge de inn i fronten av den andre listen. I dette tilfellet lar vi rekursjonen gå over den første listen alene. Reglene er:

1. Å konkatenere en tom liste med en liste gir den samme listen som resultat.
2. Å konkatenere en ikke-tom liste med en liste er det samme som å legge første element i den ikke-tomme listen inn foran resultatet av å konkatenere resten med den andre listen.

Dette kan vi kode direkte slik:

$$\begin{array}{l} \textbf{val rec}\ append = \textbf{fn}\ (nil, ys) => ys \\ \qquad\qquad\qquad\qquad\quad \mid (x :: xs, ys) => x\ ::\ append(xs, ys); \end{array}$$

Vi kan nå omgjøre *append* til en *infiks* form ved å gi kommandoen **infix** *append*. Dermed kan vi bruke *append* på nøyktig samme måte som @. Infiks-kommandoen sperrer dessuten for å omdefinere navnet. Operatoren **op** vil omgjøre en infiks operasjon til en funksjon av et par av argumenter i et uttrykk.

– **infix** $append$;

– $([1,2,3]\ append\ [4,5,6]) = ([1,2]\ @\ [3,4,5,6])$;

val $it = true : bool$

– **op** $append$;

val $it =$ **fn** $:'a\ list *' a\ list -> 'alist$

– **op** $+(2,3)$;

val $it = 5 : int$

Vi kan bruke nesten samme metode som for *append* til å lage en funksjon som avgjør hvorvidt et element er *medlem* i en liste eller ikke. Igjen må funksjonen motta et *par* som argument, der første element i paret er en liste, og det andre er det elementet vi leter etter:

val rec $mem =$ **fn** $(nil, a) => false$
$\quad | (x :: xs, a) => a = x$ **orelse** $mem(xs, a)$;

I alle eksempler hittil har vi sett på s.k. *enkel* rekursjon, d.v.s. funksjoner der evalueringen fører til at man nærmer basetilfellet skritt for skritt i en enkel sti. Ett alternativ til enkel rekursjon er *multippel* rekursjon, der evalueringen av funksjonen fører til at mange ulike stier mot basetilfellet må håndteres av kompilatoren samtidig. Som et eksempel på dette, skal vi lage en funksjon som summerer elementene i to lister parvis, og vi skal tillate listene å være av **ulik lengde**! Det betyr at dersom funksjonen mottar et par av lister som ikke er like lange, må vi kunne håndtere en *feilsituasjon*.

Feilsituasjoner kan vi kontrollere i ML ved hjelp av **exception**-setningen. En setning som

– **exception** <funksjonsnavn>

forteller ML at denne funksjonen kan føre til feilsituasjoner som vi ønsker informasjon om. Inne i definisjonen av funksjonen kan vi så bestemme når en feilsituasjon har oppstått, slik at vi får beskjed. Dette gjør vi ved hjelp av det reserverte ordet **raise**. I vårt tilfelle ønsker vi å få en feilmelding når vi finner ut at de to listene er av ulik lengde, d.v.s. når den ene er *nil* og den andre fortsatt inneholder elementer:

exception $addlist\_Error$;
val rec $addlist =$ **fn** $(nil, nil\ :\ int\ list) => nil$
$\quad | (nil, ys) =>$ **raise** $addlist\_Error$
$\quad | (xs, nil) =>$ **raise** $addlist\_Error$
$\quad | (x :: xs, y :: ys) => x + y\ ::\ addlist(xs, ys)$;

Unntakshåndtering er også nyttig når vi nytter rekursjon over heltall. Som eksempel kan vi se på den "klassiske" rekursive funksjonen $f(n) = n!$. Fakultetsfunksjonen er enkel å kode i ML:

val rec $fac =$ **fn** $x =>$ **if** $x = 0$ **then** 1 **else** $x * fac(x - 1)$;

Men ser vi på rekursjonskallet, er det opplagt at denne funksjonen aldri vil terminere dersom den gis et *negativt* heltall som argument! Dette kan vi løse ved å nytte en unntaksregel i funksjonsdeklarasjonen:

exception $fac\_Negative\_Argument$;
val rec $fac =$ **fn** $x\ =>$ **if** $x < 0$
$\quad$ **then raise** $fac\_Negative\_Argument$
$\quad$ **else case** x **of** $0 => 1$
$\quad\quad | x => x * fac(x - 1)$;

Funksjoner i ML tar alltid bare ett argument, men dette argumentet kan være et tuppel. Vi kan også definere funksjoner som tar en serie argumenter. La oss se på et enkelt eksempel:

$-$ **val** $plus =$ **fn** $(x, y : int) \;\; => x + y;$

val $plus =$ **fn** $(int * int) \; -> int$

$-$ **val** $plus =$ **fn** $x \;\; => \;\;$ **fn** $y : int \;\; => x + y;$

val $plus =$ **fn** $\; : int \; -> int \; -> int$

Den *første* av disse definisjonene sier at *plus* er en funksjon som tar *ett* argument, nemlig et *par* av heltall. Dette går også fram av typebetegnelsen som ML returnerer. Den andre definisjonen, derimot, sier at *plus* er en funksjon som tar *ett* argument (et heltall), og returnerer en *ny funksjon* som verdi, nemlig en funksjon som tar et heltall og legger dette sammen med det heltallet som den første funksjonen ble gitt som argument! Vi har altså laget en funksjon som produserer funksjoner som verdi! Dette fører til at de to funksjonene får ulik signatur. Den første må kalles med et par som argument, $plus(2, 3)$, mens den andre kan kalles "som om" den tok to argumenter: $plus\ 2\ 3$. I og med at funksjoner assosierer til høyre, kan evalueringen av denne siste anvendelsen beskrives ved følgende reduksjoner:

$$plus\ 2\ 3 \longrightarrow \;\; (\mathbf{fn}\ y => 2 + y)3 \longrightarrow 2 + 3 \longrightarrow 5$$

Funksjoner som produserer funksjoner kaller vi *høyere-ordens funksjoner*. Det samme navnet nytter vi også på funksjoner som kan ta funksjoner som argumenter, noe vi skal se eksempler på senere.

La oss ta nok et eksempel på en funksjon som produserer funksjoner, og som derfor "later som" den tar flere argumenter: Vi definerer en funksjon *max* som finner det største av to heltall:

$$\begin{array}{ll} \mathbf{val}\ max = \mathbf{fn}\ x \;:\; int \;=>\; \mathbf{fn}\ y \;:\; int \;=> & \mathbf{if}\ x > y \\ & \mathbf{then}\ x \\ & \mathbf{else}\ y; \end{array}$$

Denne funksjonen kan kalles med f.eks. $max\ 2\ 10$.

La oss nå se på funksjoner som tar funksjoner som argumenter. Et enkelt eksempel er en funksjon som tar en annen funksjon, og utfører denne *to* ganger:

$-$ **val** $twice =$ **fn** $f => \;$ **fn** $x => f\ (f\ x);$

val $twice =$ **fn** $\; : \; ('a \; -> 'a) \; -> ('a \; -> 'a)$

$-$ $twice\ square\ 5;$

val $it = 625 \; : \; int$

Vi ser at *twice* tar ett argument, f og produserer en funksjon av et argument x som er slik at f anvendes to ganger på dette. Evalueringen av $twice\ square\ 5$ er slik:

$$twice\ square\ 5 \longrightarrow square(square\ 5) \longrightarrow square(25) \longrightarrow 25$$

En særlig nyttig høyere-ordens funksjon er *map* (innebygget) som tar en funksjon og en liste, og som utfører funksjonen på samtlige elementer i listen, f.eks. kan vi skrive:

$-$ $map\ square\ [1, 2, 3];$

val $it = [1, 4, 9] : int\ list$

Fordi funksjoner er førsteklasses objekter, er det ikke vanskelig å definere *map*:

$$\begin{array}{ll} \mathbf{val\ rec}\ map = \mathbf{fn}\ f \;=>\; \mathbf{fn} & nil => nil \\ & \mid (x :: xs) => funk\ x \; :: map\ f\ xs; \end{array}$$

En funksjon som er basert på samme tankengang er *filter*, som filtrerer vekk de elementene i et liste som ikke tilfredsstiller et *predikat*:

$$\begin{array}{l} \textbf{val rec } \mathit{filter} = \textbf{fn } \mathit{pred} \;=>\; \textbf{fn } \mathit{nil} => \mathit{nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \mid (x :: xs) => \textbf{if } \mathit{pred}\ x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } x :: \mathit{filter}\ \mathit{pred}\ xs \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{filter}\ \mathit{pred}\ xs; \end{array}$$

En anvendelse som $\mathit{filter}\ \mathit{even}\ [1, 2, 3, 4]$ gir $[2, 4]\ :\ \mathit{int}\ \mathit{list}$ som resultat.

En nyttig høyere-ordens operator er operatoren for *funksjonell sammensetting*. Dersom f er en funksjon som er produsert ved å sette sammen to andre funksjoner g og h, altså at

$$f(x) = g(h(x))$$

så kan vi skrive dette i ML ved hjelp av operatoren "o" (innebygget). For eksempel er *twice* en funksjon som er resultatet av å sette sammen enhver funksjon med seg selv. Dette kan vi skrive slik i ML:

– **val** $\mathit{twice} = $ **fn** $f \;=>\; f\ o\ f;$

Nå er det lettere å definere endel selektorer for lister:

– **val** $\mathit{last} = \mathit{hd}\ o\ \mathit{rev};$

– **val** $\mathit{second} = \mathit{hd}\ o\ \mathit{tl};$

– **val** $\mathit{third} = \mathit{hd}\ o\ \mathit{tl}\ o\ \mathit{tl};$

Vi ser altså at høyere-ordens funksjoner i kombinasjon med rekursive deklarasjoner gir oss svært kraftige abstraksjonsmekanismer.