Universitetet i Oslo
Institutt for informatikk

Datatyper og eksempelstudier i Creol

Nabil Mounzir Safadi

Hovedfagsoppgave

2. mai 2005

Forord

Denne hovedfagsoppgaven er en del av en sivil ingeniør-grad ved Institutt for Informatikk ved Universitetet i Oslo. Denne oppgaven er utført ved gruppen Presis modellering og analyse.

Først og fremst vil jeg takke min veileder Olaf Owe for gode tilbakemeldinger og hjelp da jeg trengte det. Jeg vil også takke Einar Broch Johnsen for innspill og korrekturlesing, noe jeg hadde nytte av.

Til slutt vil jeg takke alle kjente og kjære for støtte underveis.

Nabil Safadi
Oslo, 2. mai 2005

Innhold

1 Innledning **1**
- 1.1 Problemstilling . . . 2
- 1.2 Mål med oppgaven . . . 3
- 1.3 Tidligere arbeid . . . 3
- 1.4 Oversikt over oppgaven . . . 4
- 1.5 Resultater . . . 4

2 Creol **5**
- 2.1 Definisjoner . . . 6
  - 2.1.1 Data . . . 6
  - 2.1.2 Grensesnitt . . . 7
  - 2.1.3 Klasser . . . 8
  - 2.1.4 Metoder . . . 9
  - 2.1.5 Imperativ kode . . . 10

3 Maude **13**
- 3.1 Funksjonelle moduler . . . 13
- 3.2 Objekt-orientert programmering . . . 19

4 Utgangspunktet **23**
- 4.1 Creol Machine Code . . . 23
- 4.2 Den abstrakte maskinen . . . 24
- 4.3 Definisjoner . . . 25
  - 4.3.1 Datatyper og verdier . . . 27
  - 4.3.2 Uttrykk . . . 29
- 4.4 Evaluering . . . 30

5 Alternativer til forbedring **35**
- 5.1 Datatyper og datauttrykk . . . 35
- 5.2 Innebygde typer . . . 38
- 5.3 Evaluering . . . 39
- 5.4 Uavhengige datafiler . . . 39
- 5.5 Subtyping i Creol . . . 41
- 5.6 Brukerdefinerte datatyper . . . 44

6 Forbedring **47**
6.1 Datatyper og datauttrykk 47
6.2 Evaluering 48
6.3 Definisjonen av funksjoner og datatyper 51
6.4 Uavhengige datafiler 57
6.5 Subtyping i Creol 60
6.6 Brukerdefinerte datatyper 62

7 Eksempler på bruk i CMC **71**
7.1 Alternating bit protocol 71
7.1.1 Grensesnitt 71
7.1.2 Klasser 72
7.1.3 Oversettelse til CMC 75
7.1.4 Eksekvering 79
7.1.5 Søk 80
7.2 Sovende barberer problemet 82
7.2.1 Grensesnitt 83
7.2.2 Klasser 84
7.2.3 Oversettelse til CMC 87
7.2.4 Eksekvering 89
7.2.5 Søk 91

8 Konklusjon **93**
8.1 Problemstillingene 93
8.2 Videre arbeid 95

Bibliografi **97**

Kapittel 1

Innledning

Denne oppgaven er en del av Creol-prosjektet ved gruppen Presis modellering og analyse ved Institutt for informatikk. Creol [1, 9, 21] er et språk som er i utviklingsfasen og er ment å kunne brukes for modellering av objektorienterte og distribuerte systemer.

Creol (Concurrent Reflective Object Oriented Language) er et forskningsprosjekt ved gruppen Presis modellering og analyse på Institutt for Informatikk. Creol er et lite, objekt-orientert språk med asynkrone metodekall. Den operasjonelle semantikken til Creol er kjørbar i Maude, ettersom Creol er definert i omskrivingslogikk i form av en interpret som tar input i CMC (Creol Machine Code) og gir en kjørbar Maude spesifikasjon. Fokuset i Creol-prosjektet er å undersøke forskjellige typer programmeringskonstruksjoner for å avklare hva som egner seg i en åpen og distribuert omgivelse, og hva som tillater enkel semantikk og enkel resonnering.

Denne oppgaven bygger på en oppgave av Marte Arnestad, "En abstrakt maskin for Creol i Maude" [10] (2003) og tar for seg styrker og svakheter ved de løsningenene som ble valgt i hennes oppgave og diskuteterer hvorvidt det kunne blitt tatt andre valg enn de som ble tatt. Her skal vi også komme inn på forbedringer av det som tidligere har blitt gjort.

I [10] var det fokus på distribusjon av meldinger og metoder. Det ble spesielt sett på vakter og sammensetningen av asynkrone og synkrone kall. Oppgaven gikk ikke inn på implementasjon av arv i CMC, og heller ikke de underliggende datatypene som finnes i Creol.

I denne oppgaven, fokuserer vi på integrasjon av datatyper og brukerdefinerte funksjoner i Creols objektorienterte distribuerte setting. Vi ser på det som tidligere har blitt gjort ut ifra eksperimentering med noen

eksempler i Creol, derav tittelen ”Datatyper og eksempelstudier i Creol”. Her fokuseres det på mangel på ønskede datatyper og funksjoner. Disse eksempelstudiene dekker prinsipielle aspekter ved distribuerte systemer.

Det underliggende språket som brukes i maskinkoden er Maude [2]. Maude er et høy-nivå formelt deklarativt programmeringsspråk basert på matematisk teori og omskrivningslogikk [32]. Maude er et forskningsprosjekt startet på Stanford Research Institute (SRI International) [4] og utviklingen av språket er nå spredt rundt omkring i verden.

Et Maude-program definerer en logisk teori. Logiske deduksjoner definert i Maude brukes til å ”regne” ut et svar. ”Utregningen” foregår ved omskrivingsregler som beskriver hvordan en tilstand kan forandre seg i et steg [28]. Maude støtter parallelle objekter og asynkrone kall/meldingsutvekslinger. Utvidelsen Full Maude støtter objekt-orientert programmering, inklusive multippel arv, men ikke redefinisjon av regler/metoder. Full Maude har ikke blitt brukt i maskinkoden, men istedenfor har det blitt definert et eget interpreterende system med multippel arv og med mulighet for redefinisjon av metoder.

1.1 Problemstilling

Hovedproblemstilling i denne oppgaven kan formuleres som følger:

Hovedproblemstilling

- Hvordan vil utvidelser i Creol påvirke den abstrakte maskinen, spesielt med tanke på innføring av brukerdefinerte datatyper og funksjoner?

Her ser vi på om de nødvendige utvidelsene er gjennomførbare i den abstrakte maskinen. Vi må passe på at typesjekkingen innad i Creol kan tilpasses innføringen av brukerdefinerte datatyper og funksjoner. Dermed dukker noe spørsmål opp som har sammenheng med relaterte problemstillinger:

- Vil vi måtte gjøre store forandringer i typesjekkingen, og i så fall, vil gamle programmer fortsatt kjøre?
- Vil vi kunne foreta ulovlige overlastinger, eller vil vi miste overlastingsinfo?
- Vil det grunnleggende i CMC kunne bevares, eller fører omveltingen til at vi også må gjøre justeringer i CMC?

Ved å ha litt mer konkrete delproblemstillinger, blir hovedproblemstillingen lettere å forstå:

Delproblemstillinger

- Hvordan kan CMC utvides med brukerdefinerte datatyper og funksjoner på en enkel og naturlig måte?
- Hvordan fungerer Creol/CMC når man utvider anvendelsesområdet til å omfatte et større spekter av datatyper og anvendelser på både de nye typene og de som er definert fra før?
- Hvordan vil innføring av subtyper, både predefinerte og brukerdefinerte kunne gjennomføres i interpreten?

1.2 Mål med oppgaven

Ønsket med denne oppgaven er å få testet Creol- og CMC-koden på et lavere nivå. På den måten kan vi finne mangler og svakheter i Creol som vi kan forbedre. Dessuten ønsker vi å finne en bedre løsning for de delene av interpreten som allerede fungerer for Creol, men er tungvindt programmert og trenger en forenkling, spesielt med tanke på å gjøre det enklere å innføre funksjoner og datatyper for brukerne av Creol.

Høy-nivå målet er å få til en bedre interpret, som gjenspeiler Creol på en enkel og elegant måte. Med Creol er målet å utvikle et rammeverk egnet for modellering av objektorienterte og distribuerte systemer. Vi ser i denne oppgaven spesielt på datatyper og innføring av nye funksjoner i Creol. Svakheten ved disse var at man selv måtte oppdatere interpreten ved opprettelse av nye datatyper eller funksjoner. Det var ingen automatikk i dette, noe vi ønsket å gjøre noe med.

1.3 Tidligere arbeid

Implementasjonen av Creol ble valgt modellert i Maude. Maude tilbyr muligheten for modellering av parallellitet, med objekter som kommuniserer med meldinger, og inneholder en modell for objektorientering, beregnet på distribuerte systemer. Hadde et annet implementasjonsspråk blitt valgt, ville vi mistet mye av denne friheten, samtidig som vi ville fått rimelig stor og omstendelig kode.

Mye av strukturen i Creol ble beholdt i CMC-koden, og etterhvert ble

også kode for arv implementert. Grensesnitt i Creol brukes ved typingsinformasjon, mens det i CMC er brukt til typeanalyse av metoder i sammenheng med arv, ettersom de inneholder informasjon man har bruk for ved arv.

All evaluering av uttrykk ble foretatt som en del av interpreten. Evalueringen måtte gjennomføres enkeltvis for hver funksjon og hver data. For hver ny funksjon vi trengte, ble omstendighetene for å lage ny evaluering stor og uoversiktlig. Dessuten ble det gjort for få forsøk med eksempel-studier og innebygde datatyper/funksjoner, noe som førte til at CMC-koden ikke fungerte for så mange eksempler.

1.4 Oversikt over oppgaven

Kapittel 2 og 3 dekker bakgrunnstoffet i oppgaven. Her presenteres de viktigste begrepene for Creol og Maude, som er de to språkene som brukes i denne oppgaven.

I kapittel 4 går vi inn på Creol, CMC og deres definisjoner. Dette kapittelet ligger til grunn for forandringer vi ønsker å utføre.

Kapittel 5 tar for seg diskusjonen av det som har blitt gjort og presenterer flere forslag til løsninger.

Ut ifra diskusjonen i kapittel 5, defineres løsningene i kapittel 6.

Kapittel 7 gir noen Creol-eksempler som baserer seg på løsningene i kapittel 6.

1.5 Resultater

Som vi vil se i kapittel 6 har oppgaven resultert i nye løsninger for Creol og interpreten, hvor vi har fått en forbedret datatype- og funksjonsløsning for Creol i CMC. I tillegg til at oversettelsen blir mekanisk, er også evalueringene forbedret deretter. Løsningene har ført til enkelhet på flere måter, blant annet ved at datatypene og funksjonene er separert fra interpreten.

Kapittel 2

Creol

I dette kapittelet ser vi på implementasjonsspråket Creol [1]. De viktigste begrepene for Creol vil bli gjennomgått her.

Creol (Concurrent Reflective Object Oriented Language) [9, 1] er under utvikling ved Universitetet i Oslo. Språket bygger på ideer fra OUN (Oslo University Notation) [26, 27] og er et formelt fundert modelleringsspråk laget for å kunne utvikle objektorienterte, åpne distribuerte systemer, som skal sikre enklest mulig analyse av pålitelighet og korrekthet av systemer. Språksyntaksen er inspirert av andre objektorienterte språk som Simula [19], Java [13] og Corba [12].

Creol er et sterk typet språk [16]. Dette begrunnes med at Creol har som mål å garantere mot typefeil ved at alle typefeil blir diagnostisert. Fordelen med dette er at programmer i Creol garanterer type sikkerhet, og vi får ingen typefeil, fordi det er krevd at alle typer skal være deklarert og all bruk skal typemessig samsvare med deklarasjonene. Metoder som er kalt på, støttes av det kallende objektet og parametrene stemmer overens med hverandre [9].

Virtuell binding av metoder er også implementert i språket. Her evalueres metodene og det returneres en prosess som er et par bestående av kode og lokale tilstander (inkludert parametre). Statisk analyse sikrer at det finnes (minst) en metode som et virtuelt kall kan bindes til (men ikke hvilken metode som velges ved kjøring). Dermed unngår man feilmeldinger under kjøring som ”method not undestood”. Metoder og attributter kan ikke slettes under kjøring, da det kan forårsake kjøretidsfeil [25].

Creol er et høynivå-språk som er ment å forener objekt-orientering og distribusjon på en naturlig måte. Asynkrone metodekall og prosessor-

slippunkter er basisen for programmerings-konstruksjoner for parallelle objekter [20].

Prosessor-slippunkter brukes til å avløse en prosess ved hjelp av en ventevakt. Altså, mens en vakt venter på å bli sann, kan andre oppgaver kjøres i mellomtiden, inntil vakten har blitt sann og oppgaven er klar for kjøring. Tiden som brukes for å vente på svar på metodekall i et distribuert system reduseres og objekter skifter mellom å være aktive og ha ikke-aktiv oppførsel.

Disse konseptene for distribuerte objekt-orienterte systemer er integrert i det objekt-orienterte språket Creol, med en enkel operasjonell semantikk, som også holder orden på kontrollen av den asynkrone meldingsutvekslingen.

Den operasjonelle semantikken til Creol er definert i omskrivingslogikk [18] og er utførbar gjennom en interpret skrevet i programmeringsspråket Maude [23]. Maude er grundigere forklart i kapittel 3.

Creol støtter at deler av programmet kan bli kompilert og lagt til et kjørende system [24]. Dette fordi et åpent distribuert system kan være resultatet av mange delprogrammer, utviklet ulike steder, på ulike tider. Nye klasser, grensesnitt og metoder kan legges til i en ny klasse og kan være avhengige av de gamle kodedelene ved arv eller ved å implementere gamle og nye grensesnitt. Den gamle koden kan bli gjort avhengig av den nye gjennom nye grensesnitt, og klasseutvidelser.

2.1 Definisjoner

Vi skal her se på Creols grammatikk og definisjoner. De vikstigste begrepene for å bygge opp kode og programmer kommer til å bli gjennomgått her. Figur 2.1 viser språksyntaksen til setningene i Creol. Setningene **if** og **while** fungere som vanlig. De andre beskrives etterhvert.

2.1.1 Data

I Creol er det en felles type, `Data`, som inneholder alle basis datatypene som brukes i språket. Slike datatyper er naturlige tall (`Nat`), strenger (`Str`), boolske verdier (`Bool`), lister (`List`) og objekt-identifikatorer (`Obj`), som kan sendes som argumenter til metodene. I tillegg er **null** en konkret verdi av `Data`. Uttrykk (`Expr`), evalueres til `Data`.

Syntaktiske kategorier.	*Definisjoner.*
l in Label	$g ::= wait \mid b \mid l? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
g in Guard	$p ::= o.m \mid m$
p in MtdCall	$S ::= C \mid C; S$
S in ComList	$C ::= \textbf{skip} \mid (S) \mid S_1 \Box S_2 \mid S_1 \vert\vert\vert S_2$
x in VarList	$\mid x := e \mid x := \textbf{new}\ classname(e)$
e in ExprList	$\mid \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
m in Mtd	$\mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}$
o in ObjExpr	$\mid !p(e) \mid l!p(e) \mid l?(x) \mid p(e; x)$
b in BoolExpr	$\mid \textbf{await}\ g \mid \textbf{await}\ l?(x) \mid \textbf{await}\ p(e; x)$

Figur 2.1: Utsnitt av syntaksen til Creol [20]

2.1.2 Grensesnitt

Definisjon 1 Grensesnitt-deklarasjon

interface F (⟨parametre⟩)
  inherits $F_1, F_2, \ldots, F_m$
begin
with G
  op $m_1(\ldots)$
    ⋮
  op $m_n(\ldots)$
asm <formula on local observable trace
      restricted to any calling object>
inv <formula on local observable trace>
where <auxiliary function definitions>
end

Grensesnitt er sentrale Creol. Grensesnittene definerer operasjoner som skal støttes i klasser som implementerer grensesnitt. I definisjonen over er F, F_1, F_2, …, F_m og G grensesnitt. Parametrene i grensesnittet kan være både verdi- og objekt-parametre, typet med henholdsvis datatyper og grensesnitt. Parametrene kan arves videre i sub-grensesnitt.

Attributtet **with**, som tar et grensesnittnavn, **G** (kogrensesnitt), angir hvem som kan kalle på metodene definert i grensesnittet. Restriksjonen ligger i at kalleren, må ha implementert kogrensesnittet, noe som gjør det naturlig å ha tomme grensesnitt, dersom der er ingen krav til kalleren. Dette kommer til nytte ved opprettelse av en kobling mellom

to objekter, hvor kun den ene skal tilby metoder. Nøkkelordet **with**, kan bare ha en tilknytning, men på grunn av arv kan man ha mange **with**-klausuler. Supergrensesnittet for alle grensesnitt er **any**. Ved å bruke dette etter **with** kan alle objekter kalle metodene i grensesnittet.

Attributtet **inv** angir en egenskap som alle som de observerte hendelsene oppfyller, **asm** angir en antagelse som beskriver oppførselen eksterne objekter må ha for at invarianten skal garanteres, mens i **where** er det definert hjelpe-funksjoner. Alle disse kan sløyfes i Creol.

Definisjon 2 Metodesignatur

metodenavn (**in** $x : X, \ldots$, **out** $y : Y, \ldots$)

Definisjonen over for metodesignaturer gjelder for både grensesnitt og klasser. Parametre som skal in og brukes i metoden, angis med **in** foran seg, mens de som skal returneres tilbake angis med **out** foran seg. Eventuelt kan parametrene sløyfes, og da sløyfes parentesene også.

2.1.3 Klasser

Definisjon 3 Klasse-deklarasjon

class C (⟨parametre⟩)
inherits $S_1(a_1, \ldots, a_p)$, $S_2(b_1, \ldots, b_q), \ldots, S_k(c_1, \ldots, c_r)$
    implements $F_1, F_2, \ldots, F_l$
begin
    var <variabel deklarasjoner>
    op $init$ $== kode - kropp$ <start tilstand med uttrykk som definerer initial-verdi>
    op run $== kode - kropp$ <startmetode>
    op $obm_1(\ldots)$ $== kode - kropp$
    ⋮
    op $obm_m(\ldots)$ $== kode - kropp$
with G_1
    op $unim_1(\ldots)$ $== kode - kropp$
    ⋮
    op $unim_n(\ldots)$ $== kode - kropp$
⋮
with G_t
⋮
end

Klasser bestemmer hvordan objekter skal se ut og hva de skal inneholde. Klassene kan arve flere andre klasser, og bruke metoder fra disse. Attributtet **Inherits** angir hvilke klasser som arves. Parametrene i en klasse kan som i grensesnittene være både verdi- og objekt-parametre. I definisjonen over er F_1, F_2, ..., F_m og G grensesnitt, mens obm_1, obm_n, $unim_1$ og $unim_n$ er metodesignaturer. Metodene kan ha lokale variabeldeklarasjoner i tillegg til kode. På den måten kan variable som bare brukes lokalt i metoden "glemmes", så snart metoden har terminert og neste gang metoden kalles, er de lokale variablene nullstilt.

Vi ser at en klasse kan implementere flere interface ved hjelp av attributtet **implements**. Klassen støtter grensesnittene som står i attributtet **implements** og må implementere metoder derfra.

Som vi ser av definisjonen over,inneholder klassen typede variable (**var**) og metode-implementasjoner (**op**). En klasse kan også inneholde en konstruktør (**init**) og en startmetode (**run**). **init** lager en passende initialtilstand ved oppretting av objekter, mens **run** aktiverer objektinstanser av klassen etter at attributtene i klassen er initialisert.

Deklarasjoner av metoder over **with**-attributtet (objektmetoder) kan bare kalles av objektet selv. Metodedeklarasjoner etter **with**-attributtet kan deles av flere objekter så lenge grensesnittene er deklarert felles for de aktuelle objektene. En klasse kan ha flere **with**-attributter med forskjellige grensesnitt.

Objekter i Creol opprettes med **new** *klassenavn(parameterliste)* og vil starte med å eksekvere metoden *init* (som initialiserer variable). Parameterlisten er den samme som i klassedeklarasjonen, og parametrene i listen tilordnes direkte til variable i klassedeklarasjonen. Etter at *init*-metoden er ferdig, vil *run*-metoden, om den finnes, bli eksekvert. Objektene kjører i parallell og har alle sin interne aktivitet, altså de prosesserer sine egne metoder, også om de blir kalt av andre objekter.

2.1.4 Metoder

Definisjon 4 Metode-kropp

metodesignatur $== lokale\ variable\ ;\ setningsliste\ .$

Metoder er i en klasse definert først som signaturer i grensesnitt, og deretter er selve metoden deklarert og skrevet ut i en klasse med imperativ kode, som består av mulige lokale variable og metode-koden.

Vi kan ha flere lokale variabler og setninger skilt med semikolon. Alle lokale variable må deklareres før setningene skrives. Slike setninger og deres sammensetning kalles programsetninger.

Definisjon 6 Lokale variable

1) var $variablenavn : type$
2) var $variablenavn : type = verdi$

hvor vi ser at vi kan velge om en variabel skal tilordnes en initial-verdi eller ikke.

Metoder trenger ikke å terminere, og alle metoder (bortsett fra **init**) kan suspenderes midlertidig. Dette fordi **init** skal sikre meningsfylt tilstand før en metode utføres på objektet og kan derfor ikke inneholde slipp-unkter.

2.1.5 Imperativ kode

I Creol angis tilordning med **:=**. Setningen

x := **new** klassenavn(nil)

oppretter et nytt objekt og tilordner x den identiteten objektet får.

Creol tilbyr muligheter for ikke-deterministiske valg. Dette angis med $S_1 \Box S_2$, hvor S_1 og S_2 er setninger eller setningslister. S_1 eller S_2 blir utført avhengig av hvem som er gyldig. Om ingen er gyldige blir regelen suspendert.

Ikke-deterministisk merge angis med $S_1 ||| S_2$. Denne evaluerer S_1 og S_2 i en rekkefølge basert på hvem av disse som er gyldig først.

Metodekall

Kall på metoden kan skje fra objektet selv eller fra andre objekter, avhengig av grensesnittet. Det finnes både synkrone og asynkrone kall.

Et synkront kall på en metode, gjør at objektet venter aktivt på avslutningen av kallet. Dette betyr at objektet ikke får utføre andre oppgaver før returen på kallet har kommet. Dette er lite gunstig i distribuerte system-er, ettersom andre oppgaver kunne vært utført av objektet uavhengig av

metodekallet.

Ved lokale synkrone kall er det viktig å utføre kallet med en gang, ettersom vi vil unngå vranglås. Hadde vi satt et lokalt kall i en prosess kø ville vi måttet vente aktivt på returen og dermed ikke kommet videre i eksekveringen.

Asynkrone metodekall fungerer på en annen måte. Disse kallene gjør at objektet kan fortsette normal eksekvering inntil returen fra kallet kommer. Selv om man får utført en programmsetning mer, venter man allikevel aktivt på returen når den trengs. Dette minner om programmering med såkalte ”fremtidsvariable” [31, 7]. Definisjon 6 viser syntaksen til de forskjellige kallene.

Definisjon 6 Synkrone og asynkrone kall

- m(e;x) lokalt synkront kall
- o.m(e;x) synkront objektkall
- l!m(e) lokalt asynkront kall med etikett
- !m(e) lokalt asynkront kall uten etikett
- l!o.m(e) asynkront objektkall med etikett
- !o.m(e) asynkront objektkall uten etikett
- l?(x) metoderetur

hvor metodenavn er m, objektuttrykk er o, innparameter er e, utparameter er x og etikettnavn (*label*) er l.

Etiketter brukes ved asynkrone kall for å holde rede på hvem som gjorde kallet. Er ikke etikett med, kan man ikke be om returverdien. Etiketter er unike og opprettes når et kall initialiseres. Denne lagres så i de lokale variablene til metoden som utførte kallet. Metoden som blir kalt tar vare på etikettveridien og sender den tilbake ved retur. Synkrone kall har derimot ikke etikett fordi disse da venter aktivt på retur.

Vakter

Definisjon 7 Vakter er konjunksjoner av:

- *wait* $\in$ **Guard** (eksplisitt avløsningspunkt)

- *l?* $\in$ **Guard**, hvor *l* $\in$ `Label`
- ϕ $\in$ **Guard**, hvor ϕ er et boolsk uttrykk over lokale og objektvariable.

Vaktene i Creol brukes til å styre prosesskontrollen i en parallell setting. En vakt lager et potensielt slippunkt for objektets prosess og prosessen venter passivt til vakten blir sann. Dette kan være med på å gjøre rekkefølgen metodene blir eksekvert i ikke-deterministisk.

Retur-vakten *l?* slår til om returen på metode-kallet med etiketten *l* har kommet. Vakten *wait* hjelper til med å få bedre prosesstyring. I aktiv kode evalueres den til å være *false*, og på den måten styrer den eksplisitt objektet til å bytte prosess. Når objektet så blir ledig igjen, vil den resterende programsetningen bli hentet tilbake og *wait* vil være endret til `true`.

await tilbyr et slippunkt i sammenheng med *wait* og vakter. **await** *l?(x)* er forkortelse for **await** *l*?;*l*?(x), hvor l er en etikett og x er en variabelliste. Her venter vi på en retur fra et kall, men ettersom vi ikke vet når det kommer brukes **await**.

await *p(e;x)* er en forkortelse for *l*!*p(e)* **await** *l?(x)*, som er en typisk form for asynkrone kall. Her er *p* er metodenavnet, *e* er en uttrykksliste med `in`-parametrene som skal brukes i metoden, *x* er en variabelliste med reurverdier og *l* er etiketten.

Kapittel 3

Maude

I dette kapittelet ser vi på språket som Creol skal implementeres i, Maude.

Maude [2, 11] er et deklarativt, høynivåspråk basert på omskrivingslogikk. Det støtter både likhetslogiske og omskrivingslogiske beregninger. Maude er et statisk, sterkt typet språk. Dette begrunnes med at Maude er et interpreterende språk.

Vi skal se nærmere på konstruksjoner og definisjoner i Maude som vi kommer til å trenge i den videre diskusjonen.

3.1 Funksjonelle moduler

En funksjonell modul (fmod) består av deklarasjoner av blant annet signaturer og en mengde likninger.

```
fmod NAT-ADD is
  sort Nat .
  op 0   : -> Nat [ctor] .
  op s   : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq 0 + M    = M .
  eq s(M) + N = s(M + N) .
endfm
```

Over har vi definert en funksjonell Maude module som heter NAT-ADD. Denne har en sort Nat og funksjonssymbolene 0 og s er deklarert til

å være konstruktører ved hjelp av attributtet $ctor$. Grunnkonstruktør-termer av sorten `Nat` kan bygges opp av 0 og s. Disse er 0, $s(0)$, $s(s(0))$ og så videre, som tilsvarer 0, 1, 2, og så videre. Vi kommer senere til å komme innpå flere typer attributter. Variabler av sorten `Nat` er deklarert med nøkkelordet `vars` og med variabelnavn M og N. `var` brukes om vi bare har en variabel av en sort.

I Maude er sorter brukt til å skille mellom forskjellige typer av verdier, som integer, boolske verdier også videre, eller med andre ord, med sorter kan vi definere datatyper. Sorter er deklarert av nøkkelordene `sort` eller `sorts`, hvor forskjellen er at `sort` deklarerer kun en type, mens `sorts` kan deklarere flere typer samtidig. Eksempler på det er

```
sort  String .
sorts Int Boolean List .
```

En deklarasjon av en sort er bare navn og angir ingen verdier. Det brukes *funksjonssymboler* for å definere verdier av hver sort. En deklarasjon av et funksjonssymbol er på formen:

```
op f : s1 ... sN  ->  s .
```

Vi ser her at det nøkkelordet `op` er brukt for å deklarere funksjonssymbol. Skal flere funksjonsymboler deklareres på en gang, brukes `ops`. Man kan bruke mixfix-notasjon der argumentets posisjon angis med `_` . Eksempel på en slik deklarasjon vises med '`_+_`'-deklarasjonen i `NAT-ADD` og her ved den boolske deklarasjonen på not, som sier at vi skal ha en boolsk verdi inn, og at resultatet også skal være boolsk.

```
op not_ : Bool -> Bool .
```

Fra `NAT-ADD`-eksempelet over ser vi at + er deklarert som et funksjonssymbol. + tar to termer av sorten `Nat` som argument og returnerer en `Nat`-verdi som resultat. Understrekene ('`_`')forteller oss hvor argumentene skal stå. Termene dette gir er $0 + 0$, $0 + s(0)$, $s(0) + s(s(0))$, og så videre. Uten understrekene, ville funksjonssymbolet vært deklarert slik i prefiks form:

```
op + : Nat Nat -> Nat .
```

Termer av sorten `Nat` blir da annerledes for +-operatoren, $+(0, 0)$, $+(s(0), s(s(0))$, og så videre.

En funksjonell modul (`fmod`) i Maude spesifiserer en fler-sortet likhetsspesifikasjon. Med fler-sortet menes at man kan ha flere sorter, eller typer.

Definisjon 1 Signatur
En fler-sortet signatur $(S, \sum)$ består av

- et sett S av sorter
- en $S^* \times S$-sortet familie $\{\sum_{w,s} | w \in S^*, s \in S\}$ av funksjonssymboler.

der S^* står for en sekvens av sorter og $\sum_{w,s}$ er et sett av funksjonssymboler med aritet w og en funksjonsverdi av sort s. Vi skriver gjerne $f : w \to s \in \sum$ for $f \in \sum_{w,s}$. Hvis w er tom, blir f ofte kalt en konstant av sort s.

Grunntermer definerer verdiene. En grunnterm er bygd av konstanter og andre funksjonssymboler på en sort-korrekt måte.

Definisjon 2 Grunntermer
Gitt en fler-sortet signatur $(S, \sum)$ kan vi definere et sett av grunntermer med sort $\{S, \mathcal{T}_{\sum} = \mathcal{T}_{\sum,s|s\in S}\}$ induktivt ved disse betingelsene:

- $\sum_{\epsilon,s} \subseteq \mathcal{T}_{\sum,s}$, hver konstant av sorten s er en grunnterm av sorten s.
- Hvis $f \in \sum_{s_1...s_n,s}$ og $t_1 \in \mathcal{T}_{\sum,s_1,...,}$ $t_n \in \mathcal{T}_{\sum,s_n,}$ og $n \geq 1$, så er $f(t_1, ..., t_n \in \mathcal{T}_{\sum,s})$, funksjonssymbol anvendt på termer av riktig sort gir en term av sort s.
- Hvert sett $\mathcal{T}_{\sum,s}$ er det minste settet som tilfredstiller betingelsene over. Det vil si, kun termer som kan bli bygget opp av konstanter og funksjonssymboler anvendt på grunntermer av riktig sort, er grunntermer.

Eksempel på en signatur(hentet fra [32]):

```
sorts t t' .
ops   a b :      -> t .
op    f   : t    -> t' .
op    g   : t t' -> t .
```

Her ser vi ut i fra definisjonen for grunntermer at a, b og $g(a, f(b))$ er grunntermer av sorten t, mens $f(b)$ er grunnterm av sort t'. Fra eksempelet `NAT-ADD` over ser vi at 0 er grunnterm av sorten `Nat`.

Grunnstenene i signaturer er konstruktørene. En konstruktør sier noe om det minste mulige elementet av en grunnterm. For eksempel for `NAT-ADD` er funksjonssymbolene 0 og s deklarert til å være konstruktører, mens konstruktør-termene 0, s(0), s(s(0)) og så videre, er bygget

opp av konstruktørene. Konstruktører i en signatur angis med $[ctor]$. Termene i `NAT-ADD` er 0, s(0), s(s(0)), 0 + 0, s(0) + 0, s(s(0)) + s(s(0)), og så videre.

En grunnterm er uten variable. For å kunne definere likninger definerer vi også termer med variable.

Definisjon 3 Termer
Gitt en fler-sortet signatur $(S, \sum)$ og et variabelsett $X = \{X_s \mid s \in S\}$, vil S-sortet settet av termer $\mathcal{T}_{\sum}(X) = \{\mathcal{T}_{\sum,s}(X) \mid s \in S\}$ være definert induktivt ved følgende betingelser:

- $X_s \subseteq \mathcal{T}_{\sum,s}(X)$ for $s \in S$; det vil si, en variabel av sorten s er også en term av sorten s.
- $\sum_{\epsilon,s} \subseteq \mathcal{T}_{\sum,s}(X)$ for $s \in S$; det vil si, en konstant av sort s er også en term av sort s.
- $f(t_1, ..., t_n) \in \mathcal{T}_{\sum,s}(X)$ hvis $f \in \sum_{s_1...s_n,s}$ og $t_i \in \mathcal{T}_{\sum,s_i}(X)$ for hver $1 \leq i \leq n$.
- $\mathcal{T}_{\sum}(X)$ er det minste S-sortet settet som tilfredstiller betingelsene over.

Ettersom vi har termer med variable, kan vi definere funksjonene brukt i signaturer. Funksjonene defineres rekursivt med en mengde *likninger*. Likningene anvendes fra venstre mot høyre og anvendelsen fortsetter inntil ingen likning lenger kan brukes.

Definisjon 4 Likninger
Gitt en fler-sortet signatur $(S, \sum)$, er en ($\sum$-) likning et trippel (X, t, t'), skrevet $(\forall X)\ t = t'$, hvor X er et S-sortet variabelsett disjunkt fra $\sum$, og t og t' er termer av samme sort, altså $t, t' \in \mathcal{T}_{\sum,s}(X)$ for en $s \in S$.

En betinget ($\sum$-) likning er ett 2(n+1)+1-tuppel $(X, u_1, v_1, ..., u_n, v_n, t, t')$ for $n \geq 1$, skrevet

$$(\forall X)\ u_1 = v_1 \wedge ... \wedge u_n = v_n \Longrightarrow t = t',$$

slik at det er sorter $s_1, ..., s_n$ i S med $t, t' \in \mathcal{T}_{\sum,s}(X)$ og $u_i, v_i \in \mathcal{T}_{\sum,s_i}(X)$ for hver $i \in \{1, ..., n\}$.

Likninger deklareres med nøkkelordet `eq`. Eksempler på likninger finnes i eksemplene `NAT-ADD` og `LIST-NAT` over. For at et en likning skal slå til, må uttrykket som bruker likningen ha match på sortene brukt i liningens venstreside. På den måten slår en likning til bare om den ”matcher”

et uttrykk. Resultatet blir det som står på høyre side av likheten i likningen. Altså, i NAT-ADD, hvor vi vil addere 0 og $s(s(0))$, vil den første likningen "matche" uttrykket vi vil regne ut, ettersom $s(s(0))$ er en term av sort Nat som representeres med M i likningen og 0 opplagt matcher 0 i likningen. Vi vil få ut som svar $s(s(0))$, som stemmer akkurat med likningen.

Nå som vi har en definisjonen på signaturer, kan vi representere lister. Under er funksjonen LIST-NAT definert. Den bruker Nat-elementer i listen, som er av sort List. Vi ser at *nil* og __ er konstruktører. Konstruktøren __ forteller oss at listen tar elementer av typen Nat og gir oss en liste av typen List tilbake.

```
fmod LIST-NAT is
  including NAT-ADD .
  sort List .
  subsort Nat < List .

  op nil : -> List [ctor] .
  op __  : List List -> List [ctor assoc id: nil] .
  op length : List -> Nat .

  var N : Nat .
  var L : List .

  eq length(nil) = 0 .
  eq length(L N) = s(length(L)) .
endfm
```

Lister må representeres med hjelp av konstruktører. En av konstruktørene sier noe om skille-elementet mellom elementene i listen (blank), mens det andre sier noe om elementene i listen.

Vi ser fra eksempelet LIST-NAT over at *nil* er en konstruktør, mens __ (blank) er listens konkatenerings konstruktør. *nil* gir oss den tomme liste. Attributtene *assoc* og *id* forklares nedenfor. Et eksempel er 's(0) s(s(0)) s(s(s(0)))' som er en liste av 1, 2 og 3.

For å unngå parentes-problemer med parseringen i Maude, kan man angi assosiosiativitets-attributtet, *assoc*. Å angi at en funksjon skal være assosiativ gjøres slik:

```
op f : s s -> s [assoc] .
```

Med dette attributtet kan vi representere funksjoner på en måte uten å måtte tenke på parenteser i forhold til elementene. Disse to eksemplene

viser assosiativitet:

$$f(f(a,b),c) = f(a,f(b,c))$$
$$X\ (Y\ Z) = (X\ Y)\ Z = X\ Y\ Z$$

Vi ser at parenteser blir unødvendig, noe som kommer nyttig med når vi skal deklarere lister.

Kommutativitet, *comm* er også et viktig prinsipp som er innebygget i Maude. Kommutativitet tillater at vi kan bytte rekkefølgen av elementer uten at resultatet blir endret. Om vi endrer __-operatoren for lister til å være *comm*, får vi definert multisett. For eksempel er to multisett med de samme elementene, men med forskjellig rekkefølge på elementene, like uansett hvordan elementene plasseres. Et annet eksempel er addering, hvor resultatet er det samme uansett hvordan tallene som skal adderes er sortert. Å angi at en funksjon skal være kommutativ gjøres slik:

```
op f : s s -> s [comm] .
```

Identitets attributter, brukes til å fjerne overflødige elementer som ikke er nødvendige å ha med i en representasjon. Et eksempel på noe vi kanskje ikke har lyst til å ha med, er konstruktøren *nil* i lister. Dette bruker vi kommandoen $id : nil$ for å markere. Dermed vil disse to listene, som følger NAT-ADD, være helt like:

$$(nil\ 0\ s(0)\ s(s(0))) = (0\ s(0)\ s(s(0)))$$

Å angi en sort med identitets-attributter gjøres slik:

```
op __ : List List -> List [ ... id: nil] .
```

Vi ser fra LIST-NAT eksempelet over at listens konkatenerings konstruktør (__) er assosiativ og har identitets-attributtet *nil*.

Attributtet *ditto* brukes ved overlastede operatorer. Siden disse må ha det samme settet av attributter, kan man skrive inn alle attributtene kun en gang, og bruke ditto-attributtet på de andre deklarasjonene. Et unntak med *ditto*-attributtet er bruken av attributtet *ctor*, som må skrives eksplisitt der man bruker den. Dermed har vi forklart de viktigste attributtene vi trenger for å lage lister i Maude.

```
op __ : List List         -> List [ctor assoc id: nil] .
op __ : MultiSet MultiSet -> MultiSet [ctor ditto comm] .
```

Attributtet otherwise (owise) er også et nyttig redskap Maude tilbyr. Dette attributtet kommer til nytte når vi har en likning som skal anvendes på et deluttrykk, hvor ingen andre likninger kan anvendes. Dette vises her:

```
op _in_ : Nat MultiSet -> Bool .

var N : Nat .
var S : MultiSet .

eq N in N S  = true.
eq N in S    = false [owise] .
```

På grunn av at Maude støtter "order-sorted" spesifikasjoner kan en sort ha subsorter. Disse fungerer slik at om s' er en subsort av s, vil hvert element av s' også være et element av s. Nøkkelordene `subsort` og `subsorts` brukes avhengig om det er ett eller flere subsorter som deklareres på en gang. Subsorter deklareres slik:

```
sorts Nat, Int .
subsort Nat < Int .
```

hvor sorten Nat er subsort av sorten Int, det vil si alle sorter av Nat også er sorter av Int. Nat er med andre ord en forfining av Int. På den måten kan man bruke uttrykk som tar Int som parameter på Nat, men ikke omvendt.

Ut ifra deklarasjonene som er gitt, finner Maude minste type. For å få en bedre typeanalyse kan man bruke flere deklarasjoner.

```
op + : Int Int -> Int .
op + : Nat Nat -> Nat .
```

Ved å ha en mindre type får vi flere mulige match av venstresider i likninger. Om for eksempel ikke Nat hadde vært en subsort av Int, ville vi ikke kunne overlaste og anvende likninger med Int-parametre for sorter av Nat. Derfor er det en fordel med å ha termer av sort Nat som kan brukes som en term av sorten Int.

Ved at høyresiden i en likning er av mindre type enn venstresiden får vi en trangere type ved reduksjon og dermed kan også flere likninger gi match. Å ha en gjentatt typing fører til at vi til slutt finner den minste typen og dermed får vi en bedre typeanalyse. Derfor blir dette gjort i Maude.

3.2 Objekt-orientert programmering

Distribuerte systemer er naturlig modellert i Maude som multisett av parallelle objekter og meldinger. Maude støtter modellering av objekt-orienterte systemer ved å tilby den predefinert pakken `CONFIGURATION`.

Denne pakken tilbyr sorter som representerer det essensielle ved objekter (`Object`), meldinger (`Msg`) og konfigurasjoner (`Configuration`), sammen med en notasjon for objektsyntaks, hvor `Object` og `Msg` er subsorter av `Configuration`.

En konfigurasjon er et multisett av av objekter og meldinger som representerer en mulig tilstand i systemet. Den tomme konfigurasjonen er representert av konstanten `none`, mens konfigurasjonskonstruktøren, `__`, tar to `Configuration`-sorter og gir et multisett av `Configuration`.

Et objekt-orientert system representeres av omskrivingsregler, som igjen representerer endring ved at man går fra en tilstand til en annen. Ved at omskrivingsreglene kan anvendes samtidig på delkonfigurasjoner, så sant de ikke overlapper, gir dette en naturlig modellering av parallellitet.

For å få til asynkronitet skal reglene bare ha ett objekt i venstresiden. Dette er direkte modellering av asynkrone distribuert meldingsutvekslng. Omskrivingsregler som involverer mer enn ett objekt i venstresiden, kalles synkrone.

Omskrivingslogikk og likhetslogikk er matematisk forskjellige. I en likhet er venstresiden og høyresiden ekvivalente, ellers hadde det ikke vært noen likhetslogikk. Likhetsrelasjonen er symmetrisk, og det fører til at den er reversibel. Da blir

```
s(M) + N = s(M + N) .
```

det samme som

```
s(M + N) = s(M) + N .
```

i likhetslogisk forstand.

Omskrivingsregler fungerer på en annen måte. Man går fra en tilstand til en annen ved at systemet utvikler seg, tilstanden forandrer seg. Her er ikke venstresiden i en regel lik høyresiden, og vi kan ikke ut i fra den samme regelen reversere det som har skjedd. På den måten kan et dynamisk system modelleres. Vi kan ikke gå tilbake, da utviklingen har gått fra en tilstand til annen ulik tilstand. Dette forteller oss at:

```
konto(''Knut'', 1000) og konto(''Knut'', 600)
```

ikke er det samme, da vi *Knut* på høyre side har tatt ut 400 kroner fra kontoen sin. Dette har ført til at vi har gått fra en tilstand hvor *Knut* har 1000 kroner til en tilstand hvor *Knut* har 600 kroner og dette viser at

disse to sidene ikke er like og dermed er dette en ikke-reversibel omskriving. Omsrivingsregelen for å gå fra 1000 til 600 kroner kan se slik ut:

```
rl [taut] : konto(X, N) => konto(X, N - 400) .
```

hvor X er navnet på kontoeieren, N er saldoen.

Vi kan også definere omskrivingsregler som er betinget. Disse angis den med crl og en betingelse som må være oppfylt for at regelen skal slå til. For at *Knut* ikke skal overtrekke kontoen sin kan vi bruke en betingelse på regelen over:

```
crl [taut] : konto(X, N) => konto(X, N - 400)
             if N > 400 .
```

Eksekvering av omskrivingsreglene gjøres ved hjelp av `rew` eller `frew`-kommandoene (henholdsvis rewrite og fair rewrite). Vi har en gitt starttilstand som omskrives i et bestemt antall omskrivingssteg eller inntil vi oppnår en sluttkonfigurasjon ved terminering. Forkjellen mellom de to omskrivingene er at `rew` fortsetter med samme del av konfigurasjonen så lenge det er mulig, mens `frew` er mer rettferdig ved at den omskriver den delen som har vært klar lengst.

Ved søk brukes Maude sin innbygde søkefunksjon, **search**. Denne simulerer flere mulige eksekveringsløp, da den søker gjennom alle tilstander ved en gitt starttilstand. Utifra denne starttilstanden kan vi se på alle mulige tilstander vi kan oppnå fra denne om mengden er endelig.

Funksjonen **search** utføres ved å oppgi starttilstand, den tilstanden vi ønsker å finne, antall tilstander vi ønsker å finne og hvor mange steg vi ønsker å utføre for å finne tilstanden. Tilstanden vi ønsker å finne er beskrevet som et mønster, og det søkes etter en tilstand som kan passe inn i dette mønsteret ved mønstergjenkjenning (pattern matching).

search [antall] $t_0 \to t_p$ **such that** betingelse .

Termen t_0 er starttilstanden, termen t_p er den tilstanden vi søker og kan inneholde variabler, mens `antall` angir hvor mange steg vi ønsker å finne og kan utelates om vi ønsker å finne alle tilstander. `betingelse` er på samme form som en betinget regel og et søk kan utføres uten denne. En term t tilfredstiller søkebetingelsen dersom t_p er en instans av t og `betingelse` holder for substitusjonen. For å bestemme hvor mange steg vi ønsker å utføre skal skal $\to$ erstattes av følgende piler:

⇒ tilstander som oppnås i nøyaktig `ett steg` fra starttilstanden t_0

⇒* tilstander som oppnås i `null` eller `flere steg`.

⇒+ tilstander som oppnås i `ett` eller `flere steg`.

⇒! tilstander som ikke kan omskrives videre, `sluttilstander`.

Kapittel 4

Utgangspunktet

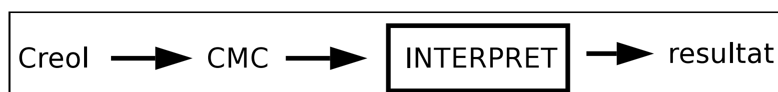
I dette kapittelet skal vi se på forskjellige aspekter ved Creol og CMC slik det var definert ved oppstart av oppgaven, som definert i [10]. Dette for å lettere kunne forstå forandringer og forbedringene som diskuteres senere. Da presenteres valgene som ble gjort, og begrunnelsene for disse.

Figur 4.1 viser hvordan vi går fra Creol-kode til et endelig resultat. Creolkoden blir oversatt til CMC-kode, CMC-koden går gjennom interpreten som er skrevet i Maude som gir oss et kjørbart utgangspunkt og kan kjøres ved hjelp av Maude-maskinen. Typeanalysen i Creol er statisk. Når vi oversetter til CMC, er typeanalysen allerede utført og vi kan anta at uttrykk og setninger er type-riktige.

Ved kjøring får vi et resultat i form av en sluttkonfigurasjon, om systemet terminerer. Dette er en indirekte og lang vei å gå for å oppnå resultater. Det ønskelige hadde vært å gå fra Creol direkte til interpreten, men fordi Maude kun tar termer av Maude-typen, må Creol oversettes til CMC, slik at vi får en konfigurasjon i Maude i form av en CMC-konfigurasjon.

4.1 Creol Machine Code

CMC står for 'Creol Machine Code' og er, som navnet sier, språket som Creols maskinkode skrives i. CMC er definert i Maude av flere årsaker. I hovedoppgaven "Kompilator fra OUN til Java" [22] ble det poengtert



Figur 4.1: Fra start til mål

flere svakheter ved å bruke Java som implementasjonsspråk og oversatt kode. Blant annet var det vanskeligheter med å få til parallellitet mellom objektene, aksess av variable (hvor man må vente aktivt på retur fra kallet) og multippel arv (ikke mulig med arv fra mer enn en superklasse). Andre implementasjonsspråk hadde også tilsvarende vanskeligheter.

Maude er et høynivåspråk som gir oss muligheten for å ha parallellitet mellom objekter og vi løser problemene vi har i andre implementasjonsspråk. Valgfriheten til å gjøre det vi har behov for i Creol er stor.

CMC er input til interpreten (kapittel 4.2) og denne eksekveres i Maude. Ettersom maskinkoden ble definert i Maude, finnes det også noen begrensninger, som vi skal se på. Creol koden blir oversatt til CMC-kode. Dette skjer manuelt eller automatisk, ved hjelp av en oversetter. CMC-koden blir så interpretert av den abstrakte maskinen.

4.2 Den abstrakte maskinen

Den abstrakte maskinen, er en implementasjon av Creol i Maude. Denne implementasjonen er prøvd å bli gjort slik at CMC-koden skal bli så lik Creol som mulig, men med uunngåelige unntak. Dette skyldes enten spesifikasjonen Maude, eller behovet for informasjon som i Creol er semantisk bundet, men som trengs eksplisitt i Maude-interpreten.

Creol-objekter, klasser, metoder og uttrykk er definert som termer i Maude. Omskrivingsregler er en viktig del av interpreten. Disse "kjører" CMC ved at de representerer forandringer. Vi går fra en tilstand til en annen så lenge vi har regler som kan anvendes.

Creol-objekter definert i Maude representeres med et attributt for den aktive koden (*Pr*) og en innebygget programkø (*PrQ*) for ventende kodebiter, som fylles på etterhvert som objektenes metoder blir kalt på og eksekveres så snart de får sjansen:

```
<_: Ob | Cl:_, Pr:_, PrQ:_, Lvar:_, Att:_, Lcnt:_>
```

hvor *Ob* angir objektnavnet som identifiserer tuppelet, *Cl* klassens navn, *Pr* inneholder programlisten, *PrQ* den ventende programkøen, *Lvar* har de lokale variablene, *Att* inneholder tilstandsvariablene til objektet og *Lcnt* er en etikettverdi som brukes ved kall.

I Maude er definisjonen av Creol-klasser representert ved attributtene *Cl*, klassens navn som identifiserer tuppelet, arvelisten angitt i *Inh*, *Att*

hvor parametrene er listet opp, *Mtds*, hvor klassens metoder er representert og *Ocnt* som er en objektteller. Tuppelet ser slik ut:

```
<_: Cl | Inh:_, Att:_, Mtds:_, Ocnt:_> :
```

Metoder i klassen er representert slik:

```
<_: Mtdname | Latt:_, Code:_>
```

hvor *Mtdname* er metodens navn og tuppel-identifikator, *Latt* består av de lokale attributtene og *Code* er programsetningene.

Metodekall blir sendt som meldinger til det kalte objektet. Meldingene er merket med identifikator for mottaker og inneholder informasjon om metodenavn og parametre:

```
invoc(O, M, D)

op comp(J)
```

hvor *O* er det kalte objektet, *M* er metodene, *D* er liste over de aktuelle innparametrene, mens *J* er returverdier.

Den eksterne køen som tar seg av meldingsutvekslingen mellom objektene er representert ved et tuppel:

```
<_: Qu | Ev:_, Keep:_>
```

hvor *Qu* er en identifikator der køen knyttes til sitt objekt og *Ev* er et multisett av metodekall og metodereturer. *Keep* er en søppelhåndterer.

Regler for parallellitet mellom objektene er innebygget i Maude-maskinen. Ferdig oversatt Creol-kode resulterer i en konfigurasjon av objekter og meldinger. Omskrivingsregler kan anvendes på to uavhengige subkonfigurasjoner samtidig på grunn av parallelliteten som er innebygget i Maude. Dog vil en Maude kjøring simulere en parallell eksekvering ved sekvensiell anvendelse av regler.

4.3 Definisjoner

En datatype er et sett av verdier og et sett av operasjoner som kan brukes til å manipulere dem [16]. I Maude er det definert flere datatyper [3], men deres bruksområdet er avgrenset i forhold til det som trengs i Creol [1], da Maude sine typer tilhører flere uavhengige pakker, mens i Creol er de subsorter av en universal type (kommer tilbake til dette i kapittel 4.3.1).

Det vi ønsker i interpreten til Creol er å binde navn til verdier, slik som i Creol. I Creol bindes en verdi til en variabel, mens i Maude er det navnet på variabelen som sier hvilken verdi det er. Denne verdien hentes ut ved evaluering (kapittel 4.4). Denne evalueringen kontrolleres av interpreten, og ikke Maude selv. Evalueringen blir som et bindeledd mellom typer i Creol og Maude-maskinen. Det er da nødvendig med en representasjon av termer i interpreten, for å angi disse verdiene.

Ettersom Creol blir interpretert av en den abstrakte maskinen, skrevet i Maude, kunne vi regne med at Creol følger de samme typekravene Maude har. Maude er et interpreterende språk, og har statisk typing og dynamisk retyping av variable på grunn av pattern matching, der forbedret type kan gi tilslag i flere venstresider.

Creol på sin side har statisk typing og ingen retyping av variable, annet enn casting av uttrykk (her: et uttrykk kan utvides til en mer generell type), fordi Creol ikke har pattern matching. Dette er lagt til grunn i språkdefinisjonen. Bindingen blir bestemt før kjøring og kan dermed ikke retypes etterpå. Unntaket ligger i verdier av subtyper som kan utvides til en mer generell type. Her er det egentlig ikke snakk om direkte retyping fra en type til en annen slags type, men fra en subtype til dennes supertype. Dette skjer fordi den alltid prøver å finne den typen som passer best til en likning.

All type-informasjon er forhåndsbestemt og vi vet hvordan typene skal oppføre seg i motsetning til dynamisk binding, der typeinformasjon bestemmes ved kjøring [16].

Creol har virtuell binding av metoder, som beskrevet tidligere(kapittel 2). Det brukes dynamisk informasjon i bindingene av metoder, da de kan retypes under kjøring, som beskrevet over. Dette er prøvd gjengitt i Maude, da vi antar type-riktighet ved oversettelse fra Creol til CMC. Informasjon om typene blir lagt inn i koden der det skal legges til.

Det finnes noen definisjoner av sorter som er nyttige å se på før vi går videre med å forklare hvordan interpreten er bygd opp. En viktig del av interpreten er hvordan navn bindes til enn verdi. Til dette formål brukes `QID`. `QID`(quoted identifier) er en modul i Maude. Denne brukes for å definere sorten *Qid*, som lager tekststrenger. *Qid* brukes som identifikator på navn til verdier i CMC og kjennetegnes av en fnutt(') foran strengen. Eksempler på *Qid* er:

```
'count
'antall
```

```
'stick
```

Istedenfor *Qid* kunne man tenke seg at strenger ble brukt for å binde verdier. Dette ville imidlertid virke upraktisk, da *Qid*-verdiene også kan være strenger.

Data er universaltypen for alle datatyper, en fellesbetegner. Alle datatypene, som blant annet strenger, integer og boolske variable ble innkapslet i *Data*.

Expr (*expression*) definerer alle uttrykk og operasjoner, inklusive datatyper, da *Expr* er supersorten til *Data*.

```
subsort Data < Expr .
```

Datatypene i *Expr*-uttrykkene evalueres for å finne riktig verdi. Når et *Expr*-uttrykk evalueres finner vi ut hva uttrykkene tilsvarer. Evalueringen foregår ved at uttrykk som er representert i *Qid* hentes fra en liste som inneholder typene og deres tilhørende verdier. Denne listen er representert ved *Subst* 4.4.

To andre viktige sorter å merke seg er *List* og *DataList*, hvor *List* er supersorten til *DataList*:

```
subsort Data < DataList Expr < List .

op __ : List     List     -> List     [ctor assoc id: nil] .
op __ : DataList DataList -> DataList [ctor ditto] .
```

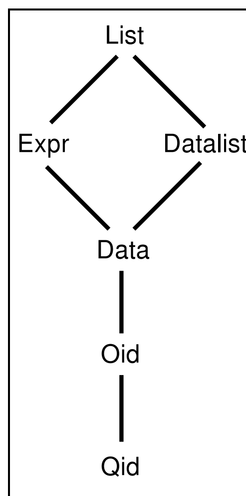
Disse brukes som skilleoperatorer i deklarasjoner og forskjellen ligger i hvilke sorter disse skilleoperatorene støtter. Dette bestemmes av hvilke subsorter disse typene har, noe vi kommer tilbake til i neste kapittel.

4.3.1 Datatyper og verdier

Creols predefinerte typevariabler må representeres i Maude. Dette gjaldt også universaltypen *Data*, som kan være boolsk, string, integer, liste eller objektidentitet.

Ettersom ønsket var å være så nær Creol som mulig med definisjoner ble løsningen å innkapsle hver av variabeltypene, ved funksjoner som tok de forskjellige typene som parametre og returnerte elementer av typen *Data*:

```
op null :       -> Data [ctor] .
op int  : Int   -> Data [ctor] .
```



Figur 4.2: Hierarki

```
op str  : String -> Data [ctor] .
op bool : Bool   -> Data [ctor] .
op list : List   -> Data [ctor] .
```

Som vi ser er typene definert som funksjoner som bruker Maude-typer i parameterlistene sine. Når det gjelder *list*, tar den sorten *List*, som har blank som skilleoperator mellom elementene i listen. Sorten *List* er supersorten til sorten *Expr*:

```
subsort Data < Expr < List .
```

På den måten kan vi fylle opp listen med elementer av sorten *Expr* i tillegg til elementer av sorten *Data*.

De andre typene er litt mer opplagte, da de svarer til Maude sine typer og **null** er en konstantfunksjon. Dermed ble representasjonen av de forskjellige typene, med eksempler, slik:

- Integer : int(1)
- String : str("hei")
- Boolean : bool(true)
- Liste : list(int(1) str("hei") bool(true))

Innkapslingen ble gjort for å unngå Maudes predefinerte typer og funksjoner. Maudes predefinerte typer ble først forsøkt brukt, men på grunn av Creols universaltype *Data*, lot det seg ikke gjøre å bruke denne som

supertype for de innebygde Maude-typene. Dette fordi Maude har predefinerte funksjoner i de forskjellige modulene sine. Disse funksjonene kan være like men ha forskjellige attributter til samme operator i et sorthierarki. Ettersom *Data* er supertypen vil det føre til at samme funksjon importeres fra forskjellige moduler og det fører til en feilmelding om at funksjonen/operatoren har blitt importert fra flere moduler med ulike attributter.

Figur 4.2 viser sortene i interpreten. Vi ser at *List* er supersorten til de andre sortene, mens *Data* også er en subsort til Expr. Ettersom *Data* er en subsort av *Expr*, utnyttes det under behandlingen av konstruktørene, noe vi skal se på i sammenheng med evaluering i kapittel 4.4.

Videre ser vi at *Oid* og *Qid* igjen er subsorter av *Data*. *Oid* står for "object identifier", og brukes her som navn på objekter etterhvert som de blir opprettet. Det som er farlig med å ha *Qid* som subsort av *Data* er at *Qid* brukes som navn på *Data*-attributter. Dette kan under evaluering (kapittel 4.4) føre til at Qid-attributter blir oppfattet som Data, noe som ikke er ønskelig. Dette skal vi komme nærmere på i kapittelet om evaluering 4.4 og hvordan det ble løst.

Qid er også farlig i forhold til *Oid*- og *Data*-sorter. Dette fordi denne brukes som identifikator til disse sortene. Vi ser av figur 4.2 at *Qid* er direkte og indirekte subsort av begge sortene *Oid* og *Data*. Dermed kan *Qid* også gi verdier av *Qid*-attributter, noe vi helst vil unngå.

4.3.2 Uttrykk

Med uttrykk var det ønsket, på samme måte som typer, at symbolene som ble benyttet i Creol også skulle brukes i CMC. Her også støtte man på problemer ettersom universaltypen *Data* kom i konflikt med Maude sine predefinerte funksjoner på samme måte som med datatypene. *Data* kunne ikke være supertypen til Maude-typene på grunn av de forskjellige modulenes egne versjoner av uttrykkene. Dermed ble det løst ved at det ble definert egne tekstlige symboler (Qid), som pluss, minus og liknende. Ved å bruke de innkapslede typene ble det dermed ingen problemer.

Overlasting av funksjoner/operasjoner var det derimot ingen mulighet for i det opprinnelige opplegget for datatyper. For eksempel operasjonen '+', der vi kunne tenke oss å bruke denne til å addere to tall sammen og å sette sammen to tekststrenger, ville ikke kunne fungere på grunn av forskjellige operander. Dette skyldes evalueringen (kapittel 4.4), hvor man måtte si hva slags operasjon det var uten å vite hva slags type attributter den tar. Løsningen ble da å ha forskjellige navn på operatorene

ved slike konflikter, selv om det i Creol skulle være mulig å overlaste operasjoner. Eksempler på operasjoner:

```
op _+_   : Expr Expr   -> Expr .
op _cat_ : Expr Expr   -> Expr .
```

Disse var en del av interpreten, men hadde ingen annen nytte enn oppbygging og evaluering av uttrykk ettersom det bare var laget likninger for eval(kapittel 4.4) av disse utrykkene. Derimot måtte man lage sine egne funksjoner når behovet var der for å gjøre det. Eksempler er `length`-operasjonen for å finne lengden av en liste, `concat` for å sette sammen to lister og `tail` som fjerner første element fra en liste:

```
op length : List        -> Int .
op concat : List List   -> List .
op tail   : List        -> Expr .
```

hvor beregningen av uttrykkene ble gjort ved hjelp av evalueringen.

4.4 Evaluering

Substitusjon (*Subst*) er en viktig del av type-evalueringen. I CMC er alle variable med i denne *Subst*-listen, som trengs for å slå opp verdier under evalueringen (kapittel 4.4) av datatypene. *Subst* er definert slik:

```
sorts Subst Pair .
subsorts Pair < Subst .

op _:_ : Qid Data -> Pair [ctor] .

op no  : -> Subst [ctor] .
op _,_ : Subst Subst -> Subst [ctor assoc id: no] .
```

der *Pair* angir navnet og verdien på attributtet, mens *Subst* gir en liste av slike.

Ved behov for beregninger trengtes også likhetslogiske funksjoner som pakket verdier av variable ut og beregnet uttrykk. Disse er ment for å få Creol til å eksekvere som ønsket i interpreten og blir utført mellom omskrivingsstegene. Disse evalueringene er viktige som en underliggende del av den abstrakte maskinen, da de slår opp verdien til en variabel med en gang det trengs fra den medfølgende *Subst*-listen.

Disse evalueringene ble brukt for å påse at all likhetslogikk gikk riktig for seg og trengtes også for alle funksjoner som ble brukt i et CMC-program. Funksjoner som blir brukt til evalueringene er:

- **val**, som finner verdien til en variabel.
- **eval**, som beregner et uttrykk og returnerer typen *Data*.
- **evalList**, som beregner verdien til hvert uttrykk i en liste og returnerer en liste med elementer av typen *Data*.

Variasjoner av den likhetslogiske funksjonen **eval** er **evalB**, som beregner et uttrykk og returnerer typen *bool*, **evalI**, som beregner et uttrykk og returnerer typen *int*, **evalS**, som beregner et uttrykk og returnerer typen *String* og **evalG** (nå kalt **enabled**), som beregner sannhetsverdien til en vakt og returnerer typen *bool*.

Disse variasjonene utfyller **eval** hvor det som skal evalueres for eksempel er en av datatypene som finnes eller en vakt (guard). Det finnes flere likhetslogiske funksjoner som ikke er en del av denne oppgaven, men kan leses i [10].

De likhetslogiske funksjonene tar imot to parametre, og returnerer et svar som er avhengig av de to parametrene. Den ene parameteren er et uttrykk (eller et variabelnavn) som man skal finne verdien av. Den andre parameteren er en liste av variable (`Subst`) som er sendt med. Denne listen traverseres rekursivt for å finne og returnere verdien til en variabel. Om variabelens verdi ikke finnes i listen returneres variabelens navn. Det kan være problemer med denne løsningen, da det kan "lure" programmereren til å tro at ting gikk bra. Om man velger en annen løsning, som å returnere null, eller en slags feilmelding, som forteller at verdien til denne variabelen ikke finnes i listen. En annen sak er at uttrykk med *Oid* som er *Qid*. For eksempel med uttrykket x = 'ole, hvor 'ole er en verdi i seg selv. Her er noen eksempler på `val`, `eval`, `evalList` og `evalI`:

```
op val   : Qid  Subst -> Data .
op eval  : Expr Subst -> Data .
op evalI : Expr Subst -> Int .
op evalList : List Subst -> List .


vars Q Q' : Qid .
vars X X' : Expr .
var  D    : Data .
var  L    : Subst .
var  C    : Int .
var  S    : String .
var  B    : Bool .
var  J    : List .
```

```
eq val(Q, no)              = Q .
eq val(Q, ((Q' : D), L)) =
          if Q' == Q then D else val(Q, L) fi .
```

Som vi ser, bruker vi `val` for å finne verdien til en variabel. Her er vi uavhengige av hva slags uttrykk det er. Det brukes Expr-uttrykk for å lette behandlingen av denne operasjonen, og på den måten får vi med oss alle *Data*-uttrykken.

Evalueringen av basale typer skjer slik:

```
eq eval(null,      L) = null .
eq eval(int(C),    L) = int(C) .
eq eval(str(S),    L) = str(S) .
eq eval(bool(B),   L) = bool(B) .
eq eval(Q,         L) = val(Q, L) .
```

Det finnes flere evalueringer for funksjoner. Eksempler på noen av disse kan sees her:

```
eq eval((X + X'), L) = int(evalI((X + X'), L)) .
eq eval((X = X'), L) = bool(eval(X, L) == eval(X', L)) .
eq eval((X > X'), L) = bool(evalI(X, L) > evalI(X', L)) .
```

Liste evalueringen foregår ved hjelp av `evalList`. Denne tar ut elementene fra en gitt liste `J` og sjekker elementene `X`, som er av typen *Expr* (supertypen til *Data* som da inkluderer *Expr*-utrykk i tillegg til uttrykk av typen *Data* og dens subtyper *Oid* og *Qid*) mot en gitt *Subst*-liste:

```
eq evalList(nil, L) = nil .
eq evalList(X J, L) = eval(X, L) evalList(J, L) .
```

Bruken av **evalList** skjer først og fremst i omskrivingsreglene der uttrykk blir evaluert, slik at vi kan være sikrere på at vi har de riktige uttrykkene. På samme måte som med eval, returneres det **nil** (for eval returneres det **null**) om uttrykket ikke finnes, noe som er tilstrekkelig da vi ikke trenger å evaluere et uttrykk som ikke brukes. Men også vanlige lister som finnes i CMC-koden trenger denne evalueringen som hjelp til `eval` når vi får inn lister:

```
eq eval(list(J), L) = evalList(J, L) .

eq evalI(Q,      L) = evalI(val(Q, L), L) .
```

For å vise et eksempel på en eval-kjøring, bruker vi en liste med to elementer, hvor hver linje er en eksekvering i Maude-maskinen:

```
eval(list(int(4) str(''hei'')), L)

list(int(4) str(''hei''))
```

L er i dette tilfellet en vilkårlig *Subst*, gjerne der den tomme substitusjonen angis som no. Som vi ser blir ikke elementene inne i listen berørt, men bare selve listen. Det er bare det utvendige som kommer deg gjennom evaluering. På den måten har vi ikke muligheten til å undersøke det som er inne i listen.

Andre hjelpefunksjoner for eval er deklarert slik:

```
eq evalS(str(S),  L) = S .
eq evalI(int(C),  L) = C .
eq evalB(bool(B), L) = B .
```

Disse brukes når det er evaluering av uttrykk, slik at vi kan bruke Maude sine predefinerte funksjoner til å regne ut disse.

Når man definerer sine egne funksjoner, må man også huske på å lage evalueringskode for funksjonene. Dette fordi å slå opp verdien til en variabel er noe man ønsker skal skje umiddelbart og derfor er det viktig å få alle variablene evaluert i en funksjon. Her er to eksempler på likhetslogiske funksjoner, pluss og cat:

```
vars X X' : Expr .

eq eval(pluss(int(X), int(X')), L) =
                  int(evalI(X, L) + evalI(X', L)) .

eq eval(X cat X', L) = str(evalS(X, L) + evalS(X', L) ) .
```

Om vi skal addere to tall, som for eksempel int(2) og int(3), går vi gjennom denne prosessen, der L er en vilkårlig *Subst*-liste:

```
eval(pluss(int(2), int(3)), L) .

int(evalI(2, L) + evalI(3, L)) .

int(2 + 3) .

int(5) .
```

Vi ser at Maude sine innebygde funksjoner blir tatt i bruk for å få til den endelige adderingen. For cat, som skal sette sammen to strenger blir også Maude sine innebygde funksjonene tatt i bruk:

```
eval(''hei'' cat ''sann'', L) .

str(evalS('hei'', L) + evalS(''sann'', L)) .

str('hei'' + ''sann'') .

str(''heisann'') .
```

Disse eksemplene viser svakheten med CMC, der ingen funksjoner blir overlastet, selv om Maude godtar overlasting. Dette skyldes evalueringen i interpreten.

Kapittel 5

Alternativer til forbedring

I dette kapittelet skal vi se på forskjellige forbedringsmuligheter og forbedringsalternativ for Creol og CMC. Forbedringspotensialet til CMC på dette punktet er stort og alternativene er mange. Her skal vi resonnere oss frem til en løsning ved å se på fordeler og ulemper ved de mulighetene vi har. Målet er å få til en felles plattform for alle funksjonene, og gjøre det enkelt å legge inn nye funksjoner i Creol og CMC.

Maude er et nytt språk i utvikling. Selv om Maude er godt etablert, er det et språk i fortsatt utvikling og med stadig nye versjoner. Derfor er det viktig at den abstrakte maskinen er med i utviklingen og utnytter det nye potensialet Maude-utviklingen kan gi, som blant annet socket support med input og output som kommer i de fremtidige Maude versjonene. Men det er ikke bare Maude som er med på å forandre den abstrakte maskinen. Svaktheter i CMC-koden samt utvidelser og forandringer i Creol er også med på å kreve en forandring.

Ettersom Creol-språket utvikles og flere tester utføres på både språket og maskinkoden, melder det seg behov for flere datatyper og funksjoner på disse som må behandles av CMC-koden. Med mange datatyper ble det fort uoversiktlig, og dette ønsker vi å finne en løsning på.

5.1 Datatyper og datauttrykk

Det vi ønsker i Creol, er abstrakte datatyper der vi selv kan definere hva de skal inneholde, og hvor vi selv bestemmer hva slags operasjoner som skal brukes på disse. Slik får vi et større rom å spille på. Blant annet støttes ikke i Maude datatyper fra forskjellige moduler i det samme settet av verdier. Med dette menes at for eksempel en liste i Maude tar kun en datatype i listen sin (sammen med denne typens subsorter). Det er på grunn av konflikt ved en del predefinerte sorter i Maude. Dette har å

gjøre med hvordan modulene er laget og hvilken hensikt de har, og ikke med språket Maude. Derfor er det spesielt viktig å definere datatyper som er egnet til Creols bruksområdet i CMC og omgå Maudes restriksjoner, noe vi løser ved å innkapsle datatypene i sorten *Data*.

Det viktige med abstrakte datatyper er at vi skiller det som er viktig (funksjonaliteten) fra detaljene (den konkrete implementasjonen). Dermed kan vi gjenbruke typene i andre programmer, og vi kan forandre kodingen av de abstrakte datatypene uten å forandre resten av programmet fordi grensesnittet er det samme [16]. Det er dette vi prøver i få til i CMC.

Datatyper i Maude defineres direkte, noe som gjør dem fleksible. Dette er høynivå deklarasjon av datatyper. Det finnes ingen pekere (med unntak av Oid - Objekt Identifier - som er referanser), dot-notasjon på variabler eller sideeffekter [32] som gjør semantikken i Maude vanskelig. Eksempler på datatyper er blant annet Int, Bool, lister og String for å nevne noen. Disse har elementer og med entydig verditype, i tillegg til funksjoner på disse.

Ettersom behovet for nye datatyper og funksjoner i CMC økte, oppsto også behovet for å gjøre det enklere å definere nye datatyper og funksjoner i CMC. Målet var å unngå å lage nye evaluerings-regler(kapittel 5.3), hver eneste gang en funksjon skulle brukes. Derimot måtte det finnes en måte å lage nye funksjoner, uten å måtte se på koden i interpreten.

I CMC er Creols datatyper definert ved konstruktører. Disse datatypene har et annet bruksområde enn de som er definert i Maude, og derfor er det behov for å gjøre dem om slik at de passer formålet vårt. En ide var skille dem fra Maude-definisjonen ved å forandre navn på dem og bruke dem som funksjoner. Dermed måtte vi definere dem som Quoted identifier(*Qid*). Da ville datatypene se slik ut:

```
'int(8)
'str(''hei'')
'bool(true)
```

Ettersom det i CMC ikke blir noen forandring ved å definere datatypene som *Qid*, eller beholde det slik det var tidligere forkastet vi denne løsningen.

Måten vi da skiller CMC sine datatyper fra Maude sine predefinerte typer er slik det har blitt gjort tidligere med innkapsling, slik at de konstruktørene vi bruker er innkapslet i typen *Data*. Konstruktører som Int, String

og lignenede defineres da ut i fra Maude-definisjonen og kalles det samme. Forskjellen er at vi må angi typen på konstruktøren som navn på en funksjon og verdien parameter.

For eksempel Int, som er av sort *Data* (som igjen er en subsort av *Expr*), defineres som tidligere:

```
op int  : Int       -> Data [ctor] .
```

Dette resulterer i at for eksempel 8, representeres ved int(8) i CMC-språket. Det samme skjer for String og Bool:

```
op str  : String    -> Data [ctor] .
op bool : Bool      -> Data [ctor] .
```

Dermed representeres for eksempel Stringen "hei" som str("hei") og den boolske variabelen true som bool(true). Dette er beholdt slik det ble definert tidligere. Alternativt, kunne vi definert 'int, 'str, 'list og liknende som Qid som tar parametre som funksjoner. Dette forkastes fordi Maude ikke forstår at disse er konstruktører, men behandler dem som funksjoner, noe som forårsaker feilmelding om flere måter å parsere en funksjon på. I Maude må vi ha konstruktører for *Data*-typer deklarert for at det skal gå i orden.

Det meldte seg også behov for å definere funksjoner på en måte som gjorde at alle fulgte samme mønster. Til det kunne vi bruke *Call*-definisjonene som var ment brukt til å opprette instanser av klasser, men gjenbrukt til funksjonskall. En *Call* er definert slik:

```
op _[_] : Qid List   -> Call [ctor] .
```

der *Call* er en subtype av CMC-uttrykket *Expr*, som definerer CMC-uttrykk og *List* er som definert tidligere en skilleoperatorer som sier noe om sammensetningen mellom elementer. *List* definerer hvordan elementene i klammeparentesen i en *Call*-definisjon skal være. Skille-elementet som er brukt er mellomrom. Vi kunne brukt komma eller andre slags skiller, men dette går bare på skjønn og et tomt mellomrom ble valgt.

Data er også en subtype av *Expr*, men *Data* er disjunkt fra *Call*, da disse kun har til felles å ha samme supertype. Det vil si at de ikke arver noe fra supertypen *Expr*, men derimot at begge disse kan brukes som *Expr*-uttrykk. I andre programmeringsspråk, som Java [13], ville for eksempel to sub-metoder ha flere felles egenskaper som de arver fra supermetoden, og dermed ikke være helt disjunkte. Men i Maude er det ikke slik.

Creol-funksjoner trenger ikke defineres som Maude-operatorer lenger.

Tidligere måtte for eksempel en funksjon som finner det største av to tall (max) defineres slik:

```
op max : Expr Expr -> Expr .

vars X X' : Expr .

eq max(X, X') = if (X >= X') then X else X' fi .
```

Nå kan dette forandres til enklerer:

```
eq 'max[X X'] = if (X >= X') then X else X' fi .
```

Vi legger merke til at X og X' er som definert over Expr og at vi nå slipper å deklarere operasjonene vi bruker.

5.2 Innebygde typer

En annen mulighet er å ikke ha med egendefinerte datatyper i det hele tatt, men istedet bruke Maude sine innebygde typer. Man kan fjerne datatypene fra interpreten og heller la programmereren bruke det han/hun trenger direkte fra prelude.maude-filen [3, 2], og deretter bruke Maude-evalueringene i interpreten. Det er selvsagt tungvint å gjøre det på denne måten, og rimelig meningsløst å ha en slik interpret uten typer. I tillegg vil det å bruke Maude sine innebygde typer stride mot Creols typekrav, da universaltypen Data som beskrevet tidligere (kapittel 4.3.1) ikke gikk an å definere ved å bruke Maude sine predefinerte typer [10].

I Maudes egen prelude.maude-fil finnes det en modul som heter META-TERM, som er bortimot det vi er ute etter. Det finnes eksempler på forskjeller og likheter, som for eksempel:

```
op _[_] : Qid GroundTermList -> GroundTerm [ctor] .
op _[_] : Qid TermList -> Term [ctor] .
```

hvor dette kan tilsvare CMCs *Call*-uttrykk, og hvor *GroundTerm* er subsort av *Term* og *Term* er en subsort av *TermList*. Dette tilsvarer CMCs *Call*, hvor vi kan ha flere uttrykk nested i hverandre. Forskjellen ligger mest i hva de forskjellige termene er. Istedenfor *List*, som er vår sort for å skille mellom elementer i en *Call*, brukes termene *GroundTermList* og *TermList*. Disse sortene alle skiller seg ut fra *List* på den måten at *List* er en liste av uttrykk (*Expr*), mens det ikke finnes noe tilsvarende i META-TERM. Det nærmeste er *TermList*, men denne er ikke definert som supersort for *Qid*, noe vi, ut i fra Creols språkdefinisjon trenger den å være, slik *List* er. Dette måtte vi i så fall ha deklarert selv. Uansett blir

det problemer med at *Qid* i META-TERM er en supersort til Constant og Variabel (altså variabel er representert av *Qid*), mens det i CMC er subsort til *Expr*. På den måten brukes den til å slå opp verdien til variabelen, noe vi ikke kan i META-TERM.

Dessuten er *Qid* en supersort for Constant og Variabel i META-TERM, noe som hadde ført til problemer i CMC. Her har vi *Data* som supersort til *Qid*, altså at *Qid* også kan brukes som *Data*-uttrykk. Dette samsvarer med Creol, og derfor vil ikke META-TERM være en tilstrekkelig god løsning å bruke for CMC-koden, selv om visse elementer der er gode å ha med. Eksempler på det er funksjonene getName og getType som finner navn og type til et element. META-TERM kunne gitt oss en stor fordel og spart oss for mye arbeid, men i hovedsak er nok META-TERM tenkt til et annet formål, og blir vanskelig å utnytte til vår fordel.

5.3 Evaluering

Tidligere var det nødvendig å lage en ny evalueringslikhet for hver ny CMC-funksjon. Dette fordi interpreten skulle kunne forstå og evaluere nye operasjoner. Dette fungerte greit, men jo flere nye operasjoner, jo flere evalueringer måtte man lage. Dermed ble interpreten uoversiktlig og unødvendig lang. Dessuten var det ingen muligheter for overlasting av operasjoner og alle datatypene måtte ha egne operasjoner som stort sett var like men fungerte bare for hver enkelt av datatypene. Vi kunne ikke separere ut datatypene fordi eval har andre parametre.

Dessuten hadde vi problemet med at vi bare fikk evaluert funksjonene på utsiden, uten mulighet til å se på innmaten. Det vi ønsker er å kunne slå opp verdiene til alle attributtene.

En løsning på disse utfordringene, er å utvide evalueringene til å se på innmaten, men det vil ikke være en forenkling av problemet. Det vil føre til flere likhetslogiske funksjoner enn de vi allerede har på den måten at for hver funksjon må det nå lages enda flere eval-ligninger. Derimot vil en innføring av en ny måte å definere funksjoner føre til en forenkling som vi skal se på i kapittel 6.

5.4 Uavhengige datafiler

Det finnes flere løsninger for hvordan vi vil behandle det økende antall datatyper og funksjoner vi hittil har fått og kommer til å få. I kapittel 5.1, snakket vi om *Call*-uttrykk, som ville for eksempel være en fin måte

å behandle funksjoner på. Spørsmålet er hvor disse funksjonene skal stå.

En måte er å beholde det slik det har vært til nå, hvor alle datatypene og deres evalueringer er innebygget i interpreten. Man kan bruke de som finnes fra før, og/eller lage nye selv. Det å lage nye selv fører til at for hver eneste ny funksjon som lages, må det i tillegg lages en tilsvarende eval-kode for akkurat denne funksjonen. Fordelen med dette er at programmereren selv bestemmer hvordan funksjonen skal se ut og hva den skal gjøre. Ulempen er merarbeidet dette medfører når man i tillegg må lage evalueringene. Alt dette krever god kjennskap til interpreten, samtidig som den forandres for hver eneste nye funksjon. Dette fører igjen til at interpreten er forskjellig hos de forskjellige brukerne og at et program som kjører hos en, ikke kjøres hos den andre uten ekstra tilleggsfunksjoner som må sendes ved siden av. Det ønskelige er å ha en enklere plattform å få til dette på, som også er mer praktisk for programmererne.

Et annet moment vi må se på når vi vil bruke funksjoner i Creol er hvordan gjøre det enklest mulig å oversette definisjoner av Creol-funksjoner til definisjoner av CMC-funksjoner. Ønsket er at CMC skal være så lik Creol som mulig, da CMC definerer den operasjonelle semantikken til Creol. På den måten er ønsket at funksjonene i Creol skal oversettes direkte til CMC og være definert slik de er i Creol. Når vi ser på de forskjellige løsningene vi har presentert for definisjonen av funksjoner i CMC, virker det vanskelig med direkte oversettelse. Først og fremst defineres funksjoner i Creol med vanlige parenteser og uten fnutt på funksjonsnavnet:

head(enListe)

Da vil en direkte oversettelse automatisk forkaste blant annet *Call-* og META-TERM-definisjonene. Men som vi har sett er det enklere for interpreten å behandle *Call*-definisjonen på mange måter, da med tanke på evalueringen og at vi slipper å tukle med interpreten hver gang vi innfører en ny funksjon. Dermed foretrekkes det å ikke oversette automatisk. Det vi beholder fra Creol er navnet på funksjonen og parameterlisten. Pakningen rundt blir litt forskjellig, men som vi har sett forenkler det mer enn det er til besvær. Dermed vil funksjonen over se slik ut når den er i bruk i et CMC program:

```
'head['enListe] .
```

En annen løsning hvor vi skiller datatypene i en egen fil fra interpreten vil også kunne være mulighet. Ettersom vi hele tiden kommer til å få flere funksjoner og typer, vil et skille mellom disse og interpreten

ha sine positive sider. Vi kunne fortsatt brukt utplukk fra META-TERM-definisjonene(kapittel 5.2), men som vi så oppfyller den ikke Creols språkdefinisjon, og å bruke deler av den vil være unødvendig resurskrevende å få til. Dessuten vil det være mer hensiktsmessig å ha en plattform å gjøre det på og ikke blande.

Denne løsningen hvor vi blander en forenkling av funksjonsdefinisjoner og egne datafiler fører til enklere å holde oss oppdaterte, og mindre programmering direkte i maskin-koden.

Den siste løsningen er en lovende metode, som vi skal se nærmere på. Her er det flere fordeler, hvor enkelthet er den største fordelen. Dessuten vil ikke maskinkoden forandre seg nevneverdig. Vi beholder strukturen ved det som tidligere er gjort.

5.5 Subtyping i Creol

I Maude er det som vi har sett, muligheter for subsorter (både predefinerte og brukerdefinerte). I Creol derimot er det ikke innebygget ennå (annet enn subgrensesnitt). Det ønskelige er om vi i Creol kunne ha subtyping. Det vi tenker på her er at typer av `Data` skal kunne ha subtyper som også er av typen `Data`. For eksempel har vi `Nat` som naturlig kan være subtype av `Int`, ettersom `Nat` bare er ikke-negative tall.

I Creol kan vi si at

Nat < Int

Problemet blir å løse det i CMC. I CMC regner vi med at typene er forhåndsanalysert og at ansvaret for å ha typene riktig ligger hos programmereren. Det vi ønsker er å ha funksjoner som bare kan brukes av subtypene og ikke supertypene.

Et eksempel som illustrerer problemet er programmet fakultet. Denne må løses ved å bruke naturlige tall, da tallet vi ønsker å beregne fakulteten av ikke kan være negativt. Er tallet negativt vil programmet kjøre til det ikke er flere ressurser å bruke, ettersom vi multipliserer tallet vi vil beregne, med tallet - 1, tallet - 2, også videre inntil vi når 1. Det sier seg selv da at vi går i en rekursjonsbrønn, om vi ikke når et basistilfelle. Slik ser fakultet ut i Creol:

class Fakultet(tall: nat)
begin

```
    var fakultet : nat = 1,
op run == beregn(tall ; fakultet)
op beregn(in n: nat, out i: nat) ==
        if (n > 2) then
            beregn(n-1 ; i);
            i := i * n
        else i := n fi
end
```

Subtype-problemet kan løses ved å ha subtyping "slått av" i CMC og interpreten. Vi kan regne med at det er gjort en gang for alle i typeanalysen, og dermed blir det som det har vært tidligere. Da vil `Nat` oversettes til `Int` når man lager CMC-kode. Eventuelt kan vi eksplisitt ha subtyper i CMC og dermed ha egne metoder som bare gjelder subtypene.

Fra tidligere i diskusjonen har vi kommet fram til at det er en fordel med overlasting av metoder, noe som er en fordel med at subsorter i CMC kan bruke supersorter sine metoder. Det som er en fordel med at det er definert egne metoder for subsorter er at vi da eliminerer muligheten til supersortene i å kunne ta i bruk metoder som er unødvendige for disse. For eksempel ved å bruke arrayer og indeksering, er det naturlig at indeksen som brukes ikke er et negativt tall. Vi kan dermed ikke bruke sorten `Int` til indeksering, men istedenfor holde oss til naturlige tall og sorten `Nat`.

Det finnes flere måter å løse dette på. Det kan være en løsning å definere en egen konstruktør for `Nat`. `Nat` vil da være en subtype av `Int`:

```
 Nat < Int .
```

På denne måten angis naturlige tall som $nat(N)$ og integer som $int(I)$, hvor N og I er variable av respektiv sorten `Nat` og sorten `Int`. Denne subtype-deklarasjonen lar seg vanskelig gjennomføre, da vi ikke får samsvar mellom funksjoner som er deklarert for `Int` og vi ønsker å bruke for `Nat`. Selv om `Nat` er en subtype av `Int` vil `Nat`-verdier ikke kunne brukes så lett. For eksempel gitt likningen

```
vars I I'    : Int .

eq 'plus[int(I) int(I')]   = int(I + I') .
```

vil ikke denne gitt tilslag om verdien er på formen $nat(N)$. Dermed må vi deklarere egne metoder for `Nat`, noe som fører til at denne løsningen er unødvendig og dermed kan den forkastes.

Å ha en konverteringsfunksjon kan også være en løsning. Denne vil virke slik at en konvertering i interpreten ser slik ut:

```
int(N:Nat) asNat = N .
```

På den måten kan vi utføre operasjoner på funksjoner med naturlige tall, og Maude sin typeanalyse seg av å undersøke at *N* ikke er et negativt tall. Problemet med dette er at vi må for hver nye subtype sette inn en slik konverteringsfunksjon i interpreten, i tillegg til at det er vanskelig å håndtere dette riktig med tanke på hvordan vi i CMC-koden skal spesifisere at det er `Nat` vi er ute etter. Da vil kanskje denne konverteringsfunksjonen som oversetter `Int` direkte til sorten `Nat` fungere bedre:

```
eq int(N) = nat(N) .
```

Her er N en `Nat`. Her er vanskeligheten igjen at `Nat` må være en konstruktørtype og nat(N) vil ikke gi match på funksjoner hvor sorten `Int` er representert.

En tredje mulighet er å bruke eksplisitt subtyping, slik det er gjort i Creol og unngår konstruktører:

```
subsort Nat < Int .
```

Altså deklarerer vi ikke `Nat`, men istedenfor bruker `Int` med en begrensing på at vi bare skal ha tall av sorten `Nat` der vi ønsker det. På den måten blir for eksempel ikke `Nat` en egen konstruktør, men en del av `Int`, med restriksjoner. Altså, de naturlige tall kan bruke alle funksjonene som er definert for `Int`, men vi kan også definere funksjoner hvor bare sorter av `Nat` godtas. Et eksempel er når vi ønsker å finne indeksen til et element i en array. Da kan vi ikke søke på indekser av negative tall. Nå har vi klart for oss fra tidligere at subsorter kan overlaste metoder av supersorten. Et forsøk på å løse subtype-problemet i CMC, er å oversette en `Nat` i Creol til en `Int`-konstruktør med `Nat` som parameter. Da blir N:Nat i Creol til int(N) i CMC, hvor N er en `Nat`. Denne metoden virker mest lovende av de vi har sett på, og dermed forsøker vi videre med denne.

Det første vi kan gjøre er å løse det ved å lage like funksjoner og sette betingelser der vi har å gjøre med subsorter:

```
vars I I'    : Int .
vars N N'    : Nat .

eq 'minus[int(I) int(I')]    = int(I - I') .
ceq 'minus[int(N) int(N')]    = int(N - N')
               if (N < N') then int(0) else int(N-N) .
```

Denne løsningen er litt farlig. Ettersom det ikke er en klar forskjell mellom typene, kan fortsatt naturlige tall overlaste funksjoner for integer med samme navn. Når vi da har to funksjoner med samme navn, en for naturlige tall og en for integer (som kan overlastes av `Nat`, da `Nat` er en subsort av `Int`), velger Maude-maskinen vilkårlig hvem som skal brukes, i fall vi har en `Nat`. Da kan svaret avhenge av hvilken som brukes, og det liker vi ikke. Dermed hjelper det ikke å lage en egen substraksjons-funksjon for naturlige tall, slik som vi kan gjøre med flytende tall, da Maude-maskinen vet hvilke funksjoner som passer til et uttrykk. Om det hadde vært en klar forskjell mellom typene (*nat(N)* og *int(I)*), kunne vi ha overlastet samme funksjonsnavn, men ikke ved subsorter i CMC. Vi kan ikke gjøre noe med svaret vi får ved å bruke en funksjon, og dermed blir det litt farlig når vi bruker for eksempel en substraksjons-funksjon.

Det blir derimot enklere å restriktere funksjoner hvor vi bare skal bruke `Nat` fra å brukes av `Int`. Eksempelet med indekser viser et problem hvor vi vil unngå typer av sorten `Int`, men istedenfor bare bruke `Nat`. Et annet eksempel er fakultets-programmet, som vi skal løsningen av i kapittel 6.5. På denne måten får vi litt bedre kontroll på funksjonene.

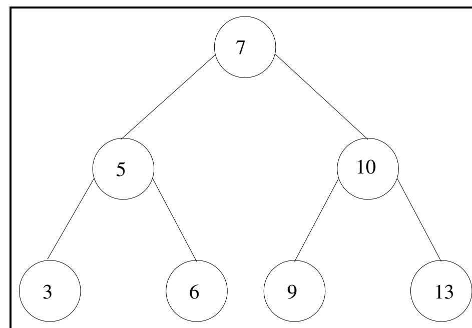
5.6 Brukerdefinerte datatyper

I Creol er det også behov får å øke antall muligheter vi har til å konstruere datatyper. For å kunne bedre mulighetene til å teste språket for andre typer, introduseres det en ny måte å lage typer på.

Disse typene skal være brukerdefinerte og skal kunne plugges inn i interpreten som vist over med uavhengige datafiler. Vi skal her undersøke hva slags datatyper vi kan forvente å kunne konstruere i Creol og hvordan vi skal løse disse datatypene i CMC og i interpreten. Spørsmålet som dukker opp er om vi klarer å være helt uavhengige fra interpreten når vi skal legge inn nye filer, eller om vi må gå inn i interpreten og gjøre noen justeringer der.

Ved å undersøke andre programmeringsspråk, vil vi kunne se flere muligheter til å kunne utnytte Creol og dennes konstruksjoner av datatyper. Ideer i de andre språkene vil kunne være til hjelp med å utvikle Creol.

Vi ser her på det funksjonelle programmeringsspråket Standard ML [17, 14, 15], hvor ML står for Meta-Language. Standard ML (SML) er en kombinasjon av LISP og Algol liknende kjennetegn og var det første språket med polymorfisk typing. Det vil si at funksjoner kan anvendes på verdier av flere typer. SML er et interpreterende språk. Her foregår interaksjonen



Figur 5.1: Tre

med SML ved å taste inn uttrykk som blir evaluert og skrevet på skjermen, ikke helt ulikt Maude.

Mye av det som tilbys i SML finnes i Creol, der i blant uttrykk, funksjoner, innbygde og brukerdefinerte typer, lister og rekursivitet. Det vi vil trekke ut fra SML er ideen med å konstruere typer som er union av to disjunkte produkter. Denne ideen går ut på å ha to sider på høyre side av likningen som er skilt med '|'. Kun en av sidene av '|' kan inneholde noe. Den andre siden må være null.

På den måten kan man for eksempel bygge opp trær på en enkel måte, hvor man på den ene siden har definert hva et blad er, og på den andre siden har man definert hva en node er. Eksempel på denne typen i Standard ML er:

```
datatype 'a tree = empty | node of 'a tree * 'a * 'a tree;
```

hvor '|' står for den disjunkte unionen av et tomt tre og et tre med noder, og '*' står for produkttypen mellom noden og dennes venstre og høyre side.

Det tomme treet *empty*, er et binært tre. Om de to trærne *tree* på høyreside av '|' er binære trær, og *'a*, som står mellom disse er av den riktige typen, så er *(tree, 'a, tree)*, et binært tre. Et blad er her representert som en node hvor begge barn er det tomme treet. Vi legger også merke til at definisjonen er rekursiv, med det tomme treet, *empty*, som basis tilfelle og at binære trær er konstruert av andre binære trær. Eksempel på et binært tre vises i figur 5.1.

I Maude er det mulig å få til noe liknende det vi ser på her. Da ved bruk av konstruktører, en for hvert tilfelle i en disjunkt union. Da vil *empty* og *tree* kunne representeres slik:

```
op empty  :              -> Tree [ctor] .
op node   : Tre Data Tre -> Tree [ctor] .
```

I Creol vil vi gjerne ha muligheten til å gjøre det samme og dermed må vi få til modellering i CMC ala produkttype og disjunkt union, slik det er i standard ML eller ved konstruktører som i Maude. Her ser vi spesielt på oppbyggingen av trær, men denne syntaksen kan brukes til å konstruere andre datatyper.

I Creol er det naturlig å definere disse datatypene slik vi har definert andre datatyper. Det vi være unaturlig å bruke syntaksen til SML og vi valgte å bruke Maudes opplegg. Det blir dessuten enklere å oversette til CMC, da vi kan følge de samme reglene som tidligere.

Det som er litt vanskelig med noder er å angi, eller ikke angi, tomme trær på venstre og høyre side. Har vi en node-struktur som forventer tre parametre, og bare skal angi et blad må vi angi venstre og høyre side med en spesiell parameter for tomme trær. På den andre siden kan vi ha en blad-deklarasjon, som forteller oss at vi har nådd slutten på treet, og på den måten slippe å angi tomme trær. Trær blir på denne måten mye enklere å lese.

Kapittel 6

Forbedring

I dette kapittelet skal vi gå nærmere innpå valgene av forandringer og forbedringer som er gjort i CMC. Som vi så i kapittel 5 var det flere aspekter ved den abstrakte maskinen som trengte fornyelse. Hovedideen er å slippe Maude deklarasjoner av nye og gamle Creol-funksjoner og ha en enhetlig syntaks på funksjonene. Dessuten vil dette føre til at vi også har en enhetlig evaluering av funksjonene. Denne likheten av formen i funksjonene gjør det enklere å regne ut verdien av utrykk.

6.1 Datatyper og datauttrykk

Call er definisjonen av funksjoner over datatyper som brukes til å konstruere funksjoner.

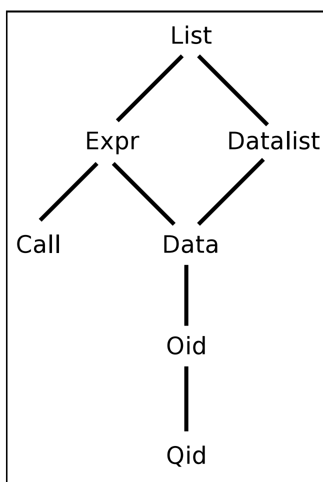
```
op _[_] : Qid List   -> Call [ctor] .
```

der det første argumentet er navnet på funksjonen, mens det andre argumentet er en liste av attributter, hvor nil angir ingen attributter.

Datatypene som bruker konstruktørene Int, String og lignende i verdimengden sin defineres på en annen måte enn konstruktørene selv. Som vi kom fram til i kapittel 5.1 følger datatypene samme konvensjon som i Maude, mens funksjoner følger en annen notasjon, nemlig *Call*-notasjonen. Fortsatt er funksjonsnavnet definert i Qid, men da med en liste av attributter i klamme-parentes, hvor *List* angir skille-elementet.

CMC-syntaksen over for *Call* er den samme for alle som vil legge til nye operasjoner i interpreten. Denne operatoren tar en Qid og innmat i form av en liste og gir en *Call* tilbake.

Elementene i *List* kan være av typen *Data* og *Expr*. Dette medfører at `bool`, `str` og `int` alle kan representeres i en og samme liste, noe som



Figur 6.1: Det nye hierarkiet

er nyttig i for eksempel definisjonen av `'pair`, som tar to vilkårlige elementer.

I CMC er sorten *Data* subsort av både *Expr* og *DataList*, *Call* er subsort av bare *Expr*, mens *Expr* og *DataList* er subsorter av *List*.

```
Qid < Oid < Data < Expr DataList < List
            Call < Expr
```

Figur 6.1 viser grafisk hvordan sortene henger sammen, fra *List* på toppen, til *Qid* helt nederst på stigen. Dette forteller oss at termer av *Data* og *Call* er disjunkte i den forstand at de ikke har felles subtype, selv om de er subsorter av *Expr* begge to. Dette er til hjelp, ettersom vi i evaluerer *Expr* i interpreten vår, og kan dermed ha flere nestede funksjoner (*Call*) i hverandre, samtidig som vi ikke mister typeinformasjon av *Data* som er byggestenene i våre funksjoner.

6.2 Evaluering

Tidligere ble evalueringsfunksjonen evalList kun brukt for å evaluere uttrykk i omskrivingsreglene og lister. Nå har den derimot fått en ny funksjon. Med inntoget av Call-uttrykkene, trengs evalList for å evaluere den innvendige funksjonslisten i funksjonene.

```
op evalList : List Subst -> DataList .

var L     : Subst .
```

```
var X     : Expr .
var I     : List .

eq evalList(nil, L)    = nil .
eq evalList(X I, L)    = eval(X, L) evalList(I, L) .
```

Vi ser at resultatene blir evaluert til DataList, mot tidligere List. Dette fordi vi forventer at alle uttrykk i en funksjonsliste er av sorten Data. Også vanlige lister og der evalList brukes i omskrivingsreglene forventer vi bare Data-uttrykk.

For å løse problemet med nye evalueringer for hver ny funksjon ble datatypene fjernet fra selve interpreten (kapittel 6.4), og alle evalueringskodene som skulle evaluere funksjoner kortet ned til en enkel kodelinje for å evaluere alle nye metoder:

```
op eval             : Expr Subst -> Data .

eq eval(Q[I], L)    = Q[evalList(I, L)] .
```

Q står for Qid, og er navnet på funksjonen med en fnutt foran, I er listen av innmat, mens L er subst-liste (ikke-repetitive lister av BndVar, som er Qid og Data).

Som vi ser kan alle funksjoner nå overlaste eval, ettersom de har de samme byggestenene og vi bruker evalList for å evaluere innmaten. Dette medfører enkelthet og mindre kode.

For å finne ut om attributtene i en liste er av riktige verdier, evalueres innholdet i listen, i tillegg til andre uttrykk som brukes utenom lister og funksjoner.

Tidligere var det slik at vi bare hadde ett eval-uttrykk for å slippe gjennom typer av sorten Data. Alle slike uttrykk gikk gjennom ved at vi brukte Maudes definisjon, otherwise (owise).

```
var D      : Data .
var L      : Subst .

eq eval(D, L)          = D [owise] .
```

Ettersom datatypene ikke hadde en egen eval, ble de ikke undersøkt for innhold. Faren ved dette er at alle slags typer av Data ble evaluert til å være riktige, selv om de ble brukt på feil sted og for å unngå at for eksempel Oid ”kommer gjennom” der de ikke skal, da QID er en subsort av

DATA[1]. Denne løsningen er enkel og elegant, men ikke tilfredstillende.

Løsningen ble å fjerne denne regelen og erstattet den med egne regler for hver eneste Data som er definert. Eksempler på dette er:

```
var L     : Subst .
var Q     : Qid .
var I     : Int .
var B     : Bool .
var S     : String .

eq eval(null,    L)     = null .
eq eval(Q,       L)     = val(Q, L) .
eq eval(int(I),  L)     = int(I) .
eq eval(bool(B), L)     = bool(B) .
eq eval(str(S),  L)     = str(S) .
```

Her er I en integer i Maude, B en boolsk variabel i Maude, S en streng i Maude, mens L er en Subst-liste. Vi ser at ved evaluering av en Qid, sender vi typen videre til val for å finne konstant-verdien til Q.

Ved å gjøre det slik, får vi ikke skilt datatypene helt fra interpreten. Ved denne løsningen må programmereren ved innføringen av nye datatyper, selv sørge for å legge inn en evaluering i interpreten. Dermed må vi fortsatt, selv om det er i liten grad, lage egne evalueringer for datatyper.

Ettersom vi kun bruker Oid som objekt-navn, blir den aldri evaluert av eval-funksjonene. Dermed kunne vi beholdt eval med owise-attributtet. Det ville også vært enklere ved introduksjon av nye Creol typer, siden vi slipper å lage en ny eval for disse.

Vi beholdt datatypene slik de var deklarert tidligere, slik at de gamle datatypene fortsatt fungerer. Konstruksjonen av nye datatyper følger et enkelt mønster. For eksempel ved introduksjonen av char og float, vil vi måtte legge til disse kode-bitene i eval:

```
var C     : Char .
var F     : Float .

eq eval(char(C), L)   = char(C) .
eq eval(float(F), L)  = float(F) .
```

Det er ikke mye arbeid mot at vi for litt mer korrekthet i forhold til å bruke owise-attributtet, selv om det bryter med ideen om uavhengighet

[1] *QID* ble senere omgjort til å være subsort av *Expr*, istedenfor *Data*. Dette gjør at faren ved å evaluere *QID* som *Data* er borte.

fra interpreten. Vi slipper å gjøre dette ved å bruke den opprinnelige evalueringsfunksjonen for datatyper, og oppnå den uavhengigheten vi ønsker.

Det som er litt negativt er at vi har mindre kontroll med evalueringen ved at Expr blir omgjort umiddelbart, istedenfor at den blir delvis evaluert på forhånd for å finne ut av om den er noe annet enn en datatype eller liste.

6.3 Definisjonen av funksjoner og datatyper

Konstruktøren `list` lager lister av sorten *List* eller *DataList* i CMC. Dette for å kunne bruke lister som Creol strukturer. `list` defineres av elementene og funksjonenene sine og bruker *List* eller *DataList* som skilleelementet mellom attributtene i listen sin. Enklere sagt, er `list` det samme som en liste i for eksempel Java [5, 13].

Lister er deklarert slik:

```
op list : DataList   -> Data [ctor] .
op list : List       -> Expr [ctor] .
```

hvor en liste med *DataList* som konkatenerings konstruktør kun kan inneholde elementer av *DataList* og dennes subtyper 6.1. En liste med *List* som konkatenerings konstruktør inneholder da elementer av typen *List* og dennes subtyper 6.1. Lister kan dermed representeres slik:

```
list(data1 data2 ...) .
```

der innmaten(data1 data2 osv) er av typen *Expr* (eller av *Data* som er en subtype av både *Expr* og *DataList*).

Følger av konstruktøren *List* er at det ikke skal være komma eller noe form for skille annet enn tomt mellomrom mellom elementene inne i listene. Dette ettersom det er deklarert at *Call* skal ta *List* inne i klammeparenteser, og *List* er deklarert med kun mellomrom.

For eksempel kan lister med tre Int elementer representeres av grunntermen slik:

```
list(int(1) int(2) int(3)) .
```

Skal vi for eksempel lage en funksjon for å hente første elementet i en liste, bruker vi skjellet over for *Call*-uttrykk og skriver:

```
vars L L'     : List .
vars E E'     : Expr .


eq 'head[list(nil)]   = null .
eq 'head[list(E L)]   = E .
```

Vi ser at det å innføre lister og det å deklarere funksjoner til lister ikke er så forskjellige fra hverandre. Navnet, innmaten og eventuelle uttrykk på høyre side er det som definerer funksjoner.

Fordelen med å deklarere datatyper på denne måten, heller enn fra slik det ble gjort tidligere, er at nå er likningene uavhengige av *eval* og *Subst* og kan legges i en egen fil. Likningene skilles fra selve interpreten. Dessuten, på grunn av evaluering slipper vi å lage nye eval-funksjoner for hver nye datatype. Dette medfører mindre arbeid, og man overlater evalueringen til interpreten. Det som er sikkert er at interpreten evaluerer metodene riktig, bare en følger reglene. Her er ett eksempel til, representert ved funksjonen `'has`, som sjekker ut om en liste inneholder et gitt element:

```
eq 'has[(list(nil)) E]   = bool(false) .
eq 'has[(list(E L')) E'] = if (E' == E) then bool(true)
                             else 'has[(list(L')) E'] fi .
```

Andre nyttige operasjoner på lister er blant annet å fjerne et gitt element (`'remove`), finne lengden på listen (`'length`), fjerne første element (`'tail`), fjerne siste element (`'rest`), eller sette inn et element i en liste, enten først (`'appendLeft`) eller sist (`'appendRight`). Funksjonene for `'remove` og `'append`-funksjonene, er slik:

```
vars L L'     : List .
vars E E'     : Expr .
vars N N'     : Nat .

eq 'appendLeft[(list(L)) E]  = list(E L) .
eq 'appendRight[(list(L)) E] = list(L E) .


eq 'remove[(list(E))   E']  = if  E == E' then list(nil)
                                          else list(E) fi .
eq 'remove[(list(nil)) E ]  = list(nil) .
eq 'remove[(list(E L)) E']  = if E == E' then
                 'remove[(list(L)) E']
               else
                'plus[list(E) ('remove[(list(L)) E'])] fi .
```

Vi ser her at det er enkelt å sette inn elementer, mens ved fjerning av elementer må vi søke i listen for å finne det rette elementet.De andre funksjonene følger samme mønster og de trengs ingen videre forklaring.

Det er ikke bare funksjoner på lister som følger denne syntaksen. Absolutt alle funksjoner, fra addering til boolske metoder følger denne notasjonen. For eksempel addering, som fungerer for både tall, strenger og lister er definert i interpreten i prefiks notasjon slik:

```
'plus[E E'].
```

Denne funksjonen tar i mot to *Expr*-uttrykk og behandler dem utifra hva slags datatype de er. Det vil si at vi nå har mulighet for overlasting av operasjoner hvor *Call*-uttrykk brukes. For eksempel har både integer, list og String operasjonen 'plus for å addere tall og sette sammen lister og tekst forholdsvis. Denne operasjonen, 'plus, tar to uttrykk(*Expr*), og ved hjelp av spesifikke likninger, finner den ut hva slags uttrykk vi har. Dermed kan svaret regnes ut:

```
eq 'plus[str(S) str(S')]       = str(S + S') .
eq 'plus[int(I) int(I')]       = int(I + I') .
eq 'plus[(list(L)) (list(L'))] = list(L L')  .
```

Det er tilsvarende for andre innfiks funkjoner, som blant annet likhetsfunksjoner, multiplikasjon og substraksjon.

I CMC er derimot funksjonene '+', '-', '*', og så videre, definert ved hjelp av innfiks notasjon. I interpreten oversettes de så til prefiks-notasjon:

```
eq E + E'  =  'plus[E E'] .
eq E - E'  =  'minus[E E'] .
eq E * E'  =  'times[E E'] .
eq E / E'  =  'div[E E'] .
```

På denne måten kan vi fortsatt bruke innfiks notasjonen på de innebygde standard typene som CMC tilbyr. Oversettelsen fra innfiks til prefiks forenkler interpreten, slik at alt blir evaluert fra den prefiks formen den er i. På den måten har vi enkel kode, og noe som er lett å følge videre ved andre definisjoner av funksjoner.

De likhetslogiske funksjonene følger samme mønster, her ved noen eksempler:

```
eq E <  E'   =  'less[E E'] .
eq E <= E'   =  'lessEq[E E'] .
eq E >  E'   =  'less[E' E] .
```

```
eq E >= E'   =  'lessEq[E' E] .
eq (E = E')  =  'equal[E E'] .
eq E /= E'   =  'not['equal[E E']] .
```

Vi ser at vi gjenbruker kode ved flere av funksjonene. Det er både mer effektivt og gjør koden enklere og lettere å holde orden på. Når det blir oversatt til prefiks notasjon, bruker vi Maude sine predefinerte typer for å få til utregningen. Bortsett fra Maude funksjonen '==', kan vi ikke sjekke generelle *Data*-uttrykk ved å bruke Maude sine predefinerte likhetsfunksjoner. Dette har sammenheng med at vi i CMC har overskrevet disse funksjonene og dermed må gå helt ned til "beinet", ved å bruke Maude-typene for å finne ut av om disse typene i likhetslogikken.

```
eq 'equal[D D']                  =  'bool(D == D') .
eq 'less['int(I) 'int(I')]       =  'bool(I < I') .
eq 'lessEq['int(I) 'int(I')]     =  'bool(I <= I') .
eq 'less['str(S) 'str(S')]       =  'bool(S < S') .
eq 'lessEq['str(S) 'str(S')]     =  'bool(S <= S') .
```

Andre innebygde innfiks-funksjoner som blir oversatt til prefiks:

```
eq E cat E'  = 'plus[E E'] .
eq E and E'  = 'and[E E'] .
eq E or  E'  = 'or[E E'] .
eq E .fst    = 'fst[E] .
eq E .scd    = 'scd[E] .
```

Vi ser at cat er det samme som $'plus[str(S) str(S')]$, ettersom de begge tar to strenger og gir dem videre til $'plus[EE']$. Funksjonene $.fst$ og $.scd$, brukes for å hente forholdsvis første og andre element fra en $pair$.

Grunnen til forskjellen mellom funksjonene og byggestenene (list, int, osv) er at Maude ser på byggestenene som lister om vi bruker Call-syntaksen på disse. *Call* er også en subtype av *Expr*, akkurat som *Data*, men *Call* og *Data* har ingen felles subtype. Selv om begge er subsorter av *Expr*, regner Maude dem som disjunkte mengder. Man kunne si at vi kunne brukt den samme operasjonen på `int` og `bool`, som bare får inn en parameter, som jo er det samme som en liste med en parameter.

Dette vil mot ønske vårt være feil, ettersom vi da måtte la *Data* utgå og det ville føre til at interpreten ikke samsvarer med Creols rammeverk. Dessuten ville det også vært mulig å ha flere verdier i disse elementene, slik at de ville blitt representert som for eksempel int[int[int[...] ...] ...], noe som ikke ville gitt mening for en Int. Dessuten er det viktig for evalueringen og selve interpreten at vi kan gjenkjenne *Data* i typingen, ellers ville vi kunne få resultater vi ikke ønsker, som beskrevet over.

Datatypene i CMC-koden, *Data* og *Call*, er som beskrevet subtyper av *Expr*. Alle datatyper er av sorten *Data*, som er en subsort av *Expr*. `list` tar både typer av sorten *Expr* og subsorten *Data*.

Ved at vi kan bruke både *Data*-uttrykk og *Expr*-uttrykk (noe som fører til den tredje muligheten; *Call*-uttrykk), får vi overlasting på operasjonen *Expr*. Dette kalles ad-hoc overlasting [28], grunnet at *Call* og *Data* er subtyper av *Expr*, men allikevel forskjellige. Spørsmålet er om det er grunn til å bekymre seg for det. Vi kan komme til et punkt hvor vi har en liste inne en liste, og dermed et *Call*-uttrykk inne i en annen *Call*.

```
Qid[...[Qid[...]...]...]
```

Dette må kunne evalueres til å være riktig, fordi det ikke skal være noe i veien i å ha to eller flere *Call*-uttrykk inne i hverandre. Det er ikke ulovlig i forhold til at *Call* også er *Expr*. Dermed kan vi bruke nøsting i funksjonene våre. Et eksempel er denne matematiske utregningen, hvor addere to uttrykk som først selv må evalueres og regnes ut ved hjelp av multiplikasjon:

```
red 'plus[('times[int(8) int(1)]) ('times[int(7) int(2)])] .
```

Resultatet vi får er

```
result Data: int(22)
```

som er helt riktig.

Istedenfor å bruke sorten *List* kunne vi brukt sorten *DataList* for å skille mellom elementer. *Data* er en subsort av *DataList*, mens *Expr* ikke er det. Sorten *DataList* tar dermed ikke *Expr*-uttrykk som en del av listen sin, og dermed oppstår et problem om vi ønsker å bruke *Expr*-uttrykk i *Call*-uttrykkene våre. *DataList* vil ikke kunne ta i mot slike uttrykk, og dermed går vi i stå. Derfor er det best å holde seg til *List* som er supersorten til både *DataList* og *Expr*.

Ved å bruke *List*, mister vi informasjon som vi kanskje kunne trenge med hensyn på hva slags input vi får inn, og dessuten overlastes *Call* ved at vi kan ha *Call* nested inne i *Call*. Derfor kunne *DataList* være et alternativ, men som vi så over, ikke et godt nok alternativ.

Nå som vi har definisjonene klare kan vi se på et eksempel med evaluering av en funksjon. Funksjonen er uttrykket vi så på over, hvor vi adderer to uttrykk som først skal multipliseres. Hver linje er ett ledd i utregningen:

```
'plus[('times[int(8) int(1)]) ('times[int(7) int(2)])] .

eval('plus[('times[int(8) int(1)])
                        ('times[int(7) int(2)])], L) .

'plus[evalList(('times[int(8) int(1)])
                        ('times[int(7) int(2)]), L)] .

'plus[eval('times[int(8) int(1)], L)
                 evalList('times[int(7) int(2)], L)] .

'plus[('times[evalList(int(8) int(1), L)])
      eval('times[int(7) int(2)], L) evalList(nil, L)] .

'plus[('times[eval(int(8), L) evalList(int(1), L)])
                 ('times[evalList(int(7) int(2), L)])] .

'plus[('times[int(8) eval(int(1), L) evalList(nil, L)])
        ('times[eval(int(7), L) evalList(int(2), L)])] .

'plus[('times[int(8) int(1)])
    ('times[int(7) eval(int(2), L) evalList(nil, L)])] .

'plus[int(1 * 8) ('times[int(7) int(2)])] .


'plus[int(8) int(2 * 7)] .


'plus[int(8) int(14)] .

int(8 + 14) .

int(22) .
```

Resultatet vi får er riktig. Vi legger merke til at alle datatyper blir evaluert før vi begynner å beregne resultatet.

Andre funksjoner som er nyttige i CMC, er list- og pair-funksjonene. Deres innmat blir enkelt evaluert ved hjelp av eval og evalList:

```
vars X X'    : Expr .
var  J       : List .
var  L       : Subst .
```

```
eq eval(list(J), L)      = list(evalList(J, L)) .
eq eval(pair(X, X'), L) = pair(eval(X,L), eval(X',L)) .
```

Som vi ser, sendes evalueringen av innmaten til list videre til evalList, som igjen fordeler evalueringene til eval. Evalueringen av pair er enklere da det kun er to attributter innvendig som skal evalueres.

Evalueringen gjør at vi sjekker hver forekomst av elementene i en funksjon om den er lovlig eller ikke. Vi går utenfra og inn og sjekker om det er riktige elementer i forhold til ytterste element.

For eksempel, om vi sender inn list(int(4) str("hei")), evalueres den på følgende måte, hvor hver linje er ett ledd i utregningen:

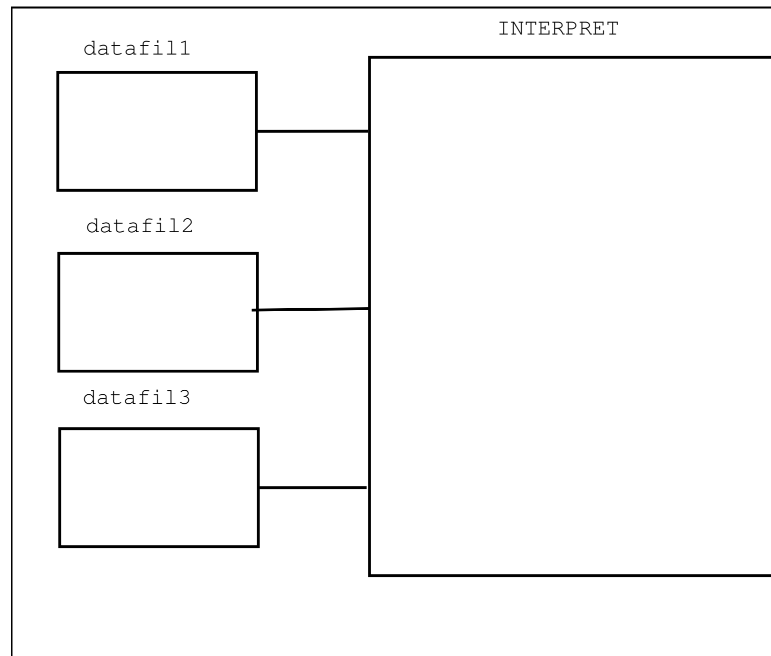
```
eval(list(int(4) str("hei")), L)

list(evalList(int(4) str("hei"), L))

list(eval(int(4), L) evalList(str("hei"), L))

list(int(4) eval(str("hei"), L) evalList(nil, L))

list(int(4) str("hei") nil)

list(int(4) str("hei"))
```

der L i dette tilfellet er en vilkårlig Subst (gjerne der den tomme substitusjonen som angis med no). Vi ser at listen vi får ut til slutt, er den vi startet med, og dermed er det en vellykket evaluering.

6.4 Uavhengige datafiler

Som vi kom fram til i forrige kapittel, vil en løsning med egne datafiler være fordelaktig. For at interpreten skal bli mer oversiktlig, ønsker vi å skille ut det overflødige som ikke har noe direkte tilknytning til interpreten i en egen datafil. Det vil si at alle evalueringene, reglene og Creolkode-evaluering foregår som før i interpreten, mens behandling av datatyper for alle elementer og deres hjelpefunksjoner utføres i en egen datafil. Med dette menes med andre ord at behandling av lister, boolean uttrykk, integer og andre datatyper settes over i en annen datafil som kjøres i tilknytning med interpreten. Det er også på grunn av dette evalueringen forenkles (kapittel 6.2).

Ideen med uavhengige datafiler stammer fra ønsket om selv å kunne



Figur 6.2: Ideen med uavhengige datafiler

tilføre Creol de datatypene og deres operasjoner en selv trenger uten å programmere direkte på koden i interpreten. Det gjør det mye enklere for programmereren å vite hvor de nye datatypene skal settes inn og programmereren velger selv hvordan disse skal brukes.

Det finnes flere muligheter for hvordan man vil implementere denne metoden. Man kan for eksempel lage et bibliotek der man kan plukke ut de datatypene en trenger som pakker, slik det er gjort i Java [5]. Eller det kan være opp til programmereren å lage sine egne datatyper og implementere dem i interpreten ved hjelp av datafiler.

Det ville vært mer praktisk og stabilt å bruke et felles filområde med de viktigste funksjonene og datatypene, lik et bibliotek. Da ville vi vært mer sikre på at datatypene er godt testet og følger retningslinjene og strukturen for hvordan datatyper deklareres. På den andre siden kan vi ikke være sikre på å oppfylle alle programmererens behov for funksjoner, noe som fører til at han selv må definere funksjoner. Derfor vil en blanding av bibliotek og mulighet for å lage en egen datafil med ens egendefinerte funksjoner være det optimale. Nye bibliotek moduler kan innarbeides etterhvert som behovet er der. Dette fører til at programmereren beholder friheten, samtidig som vi får litt struktur. Uansett hvordan det gjøres

viser figur 6.2, hvordan det er tenkt at datafilene skal flettes inn i interpreten.

Det finnes begrensninger på hvordan nye funksjoner skal lages, som beskrevet i 6.1. De må følge visse retningslinjer, men dette er til for å forenkle og for å slippe å deklarere nye spesifikasjoner hver gang en trenger det. Det er bare å gå rett på koden for det en har lyst til å programmere.

Formålet med å ha uavhengige datatyper er å kategorisere og gjøre oppdateringen lettere. For eksempel ved bruk av lister eller par, oppstår det gjerne behov for å ha med noen ekstra funksjoner. Slike kan programmereren selv lage, slik at de passer formålet.

En trenger ikke stoppe med kun en datafil (eller importere kun en fil fra biblioteket). Det er fullt mulig å ha flere filer som man kan flette inn ved behov, eller man kan sette inn alle filene samtidig for å ha flest mulig opsjoner på en gang.

Det er naturlig å la det finnes en default-datafil, ved bruk av interpreten. Denne inneholder lister, integer, strenger, booleanske uttrykk og par, samt diverse operasjoner på disse. I mange sammenhenger er dette nok (men ikke i alle).

Formålet med uavhengige datafiler er åpenbart; det gjør det mye enklere for de som holder på med Creol å holde styr på interpreten, samtidig kan alle andre slippe å kode i selve interpreten. Det er også viktig å nevne at interpreten blir mer robust, ved å gjøre det på denne måten.

På den andre siden, kan ikke en programmerer bare lage et program og kjøre det uten å ha en datafil. Interpreten uten en slik fil er ”tom”, og dermed avhengig av datafilen for å gi mening. Det vil si, i alle programmer er det behov for datatyper som boolean, int, list og lignende, samt operasjoner på disse. Dette medfører merarbeid for programmereren, som kan gå utenfor programmets problemstilling. Det at interpreten ikke er komplett med datatyper eller noen innebygde moduler, fører til at programmeren må ha kunnskaper om interpreten for å få gjort noe og for å lage sine egne funksjoner, men i mindre grad enn om det hadde vært gjort med innebygget default datatyper for interpreten. Det vil være til hjelp, selv om vi ikke kan forutse alle datatyper.

6.5 Subtyping i Creol

Ved at vi har valgt å ha typeanalyse for subtyper i Creol slått av i CMC, må vi tilpasse subtypene til interpreten på en annen måte. I kapittel 5.5 kom vi fram til at løsningen som virker best når det gjelder å oversette subtyper fra Creol til CMC blir å bruke supersorten i CMC med subsorten som parameter. Med andre ord vil en verdi av en type av en subsort i CMC være representert ved `type(K)`, hvor `type` sier hvilken supertype det er, mens **K** er en parameter av subsorten til `type`. På den måten kan vi eliminere muligheten for supersorten å bruke metoder som er beregnet på subsorten, i tillegg til at supersort-funksjoner kan overlastes av subsorten.

Et enkelt eksempel er `Int` og `Nat`, hvor `Int` er supersorten til `Nat`. En `Int` er alle tall, både positive og negative, mens en `Nat` kun er positive tall og tallet 0. `Nat` er en naturlig subsort av `Int` og vi lager noen enkle funksjoner som er spesielt tilpasset for `Nat`, da vi vil unngå å jobbe med negative verdier.

Svaret blir at det er opp til typeanalysen å holde orden på sorter og subsorter ved funksjonskall, og la Maude-maskinen ta seg av kall på funksjoner hvor man sender med parametre av sorten `Int`, men som bare skal ta parametre av sorten `Nat`.

Andre eksempler på funksjoner som bare bruker naturlige tall, kan være å finne et element på en gitt indeksplass i en array (`'index`), eller å finne alle elementer etter en gitt posisjon i en array (`'after`).

```
eq 'after[(list(E L)) int(0)] = list(E L) .
eq 'after[(list(E L)) int(N)] =
                     'after[(list(L)) (int(N) - int(1))] .

eq 'index[(list(nil)) int(N)] = null .
eq 'index[(list(E L)) int(N)] = if (N == 1) then E else
                         'index[(list(L)) int(N - 1)] fi .
```

hvor E, L og N er som angitt tidligere. Vi ser fra begge eksemplene at vi trekker fra $int(1)$ på høyre side av likningen, noe som gjør at vi kan komme i en situasjon hvor vi ser etter en plass i arrayen som er negativ. Det er to måter å unngå at dette skjer. Den ene er at vi har regler som gir et svar når vi har nådd $int(0)$ eller list(nil). Den andre er at vi vet at den innebygde typeanalysen tar seg av de negative tallene når de forekommer der de ikke skal. Siden alle venstre sidene krever naturlige tall, som i `index`-eksempelet, vil negative indekser ikke gi noen reduksjon og

evalueringen stopper opp. Slike uttrykk som ikke kan evalueres, representerer feil verdier.

I kapittel 5.5 lagde vi fakultetsfunksjonen. Nå som vi vet hvordan vi skal løse det i CMC, kan vi oversette direkte.

```
< 'Fakultet : Cl | Inh: nil,
  Att: ('tall : null), ('fakultet : int(1)),
  Mtds: < 'beregn : Mtdname |
        Latt: ('n : null), ('i : null),
        Code: if ('n > int(2))
                 th ('beregn('n - int(1) ; 'i)) ;
                    ('i := ('i * 'n))
              el
                 ('i := 'n)
              fi ;
        end ('i)
        > *
        < 'run :  Mtdname |
          Latt: no,
          Code: 'beregn('tall ; 'fakultet )
        >,
 Ocnt: 0
>
```

Vi legger merke til at klassen $'Fakultet$ tar i mot parameteren som skal beregnes i det første elementet i attributtlisten, $'tall$, mens metoden $'beregn$ har inn- og ut-parametre i den lokale attributtlisten sin, $'n$ og $'i$.

Her er det ikke angitt direkte at tallene vi ønsker å bruke skal være naturlige, slik det er i Creol, men oversettelsen fra Creol, som skal gjøres mekanisk, skal sørge for det. Kallet med på klassen $'Fakultet$ gjøres med for eksempel $'tall$ = int(5) og oppretter ett objekt av denne klassen:

```
(new 'Fakultet(int(5))) .
```

Dette eksempelet er kjørt i interpreten og beregningen foregår som ønsket, der resultatet er int(120), noe attributtet $'fakultet$ viser her:

```
< 'Fakultet0 : Ob |
   Cl: 'Fakultet,
   Pr: continue(1),
   PrQ: (await 1 ? ; 1 ?(nil)), no,
   Lvar: ('caller : 'Fakultet0),'label : int(1),
```

```
   Att: ('this : 'Fakultet0),('tall : int(5)),
                           'fakultet : int(120),
Lcnt: 6 >
```

Vi skal i kapittel 7 gå nærmere inn på flere eksempler og oversettelser, slik at det blir klarere med regler for dette.

6.6 Brukerdefinerte datatyper

Ved å undersøke et annet funksjonelt språk, Standard ML, kunne vi trekke ut ideer derfra og implementere disse i Creol. Et eksempel som vi kunne få bruk for er tre-strukturen. Som vi så tidligere var den implementert slik i SML:

```
datatype 'a tree = empty | node of 'a tree * 'a * 'a tree;
```

I Creol vil definisjonen være slik det er for andre datatyper i Creol.

op empty : -> Tre
op tre : Tre Data Tre -> Tre

forteller oss at en (rot)node er en type av sorten `Data`. `Tre` skal også da være en del av den felles datatypen Data og dette må også deklareres i Creol, som i Maude.

Vi legger merke til at deklarasjoner av datatyper og funksjoner er den samme i Creol. I Creol har vi definisjonen av trær som over. En funksjon som kan anvendes på trær er `put`, som setter en ny node i treet. Denne deklareres slik i Creol:

op put : Data Tre -> Tre

Ved en automatisk oversettelse til CMC, vil det være vanskelig å skille mellom deklarasjonene. Vi må også huske på at funksjoner ikke oversettes til deklarasjoner i CMC, men kalles på direkte ved $'f[...]$, hvor f er funksjonsnavnet. Et forslag er å bruke et attributt på slutten av en datatype som sier at den er en konstruktør. Dette er brukt i Maude i sammenheng med attributtet `ctor`.

For å modellere datatypen tre i CMC, har vi laget egne sorter og deklarasjoner for trær.

```
sort Tre .
subsort Tre < Data .
```

Vi ser her at et tre er en subsort av Data. Dette må til for at vi skal kunne bruke trær som data-uttrykk. Dermed kan vi nå uttrykke trær på denne måten:

```
op tom  : -> Tre [ctor] .
op tre  : Tre Data Tre -> Tre [ctor] .
```

hvor tom er det samme som empty, og node er representert som en Data, mens treet er representert med en rotnode (Data) med venstre og høyre subtre. Istedenfor empty er det valgt tom fordi empty er i CMC definert som en tom ProgList (*Pr*) (kapittel 4.2). Finnes ikke venstre og høyre subtre angis de med tomme subtrær. Om en node bare har et barn, må vi angi at barnet på andre siden er tomt. Slik representeres da trær:

```
tre(tom, int(8), tom) .

tre(tom, int(8), tre(tom, int(9), tom)) .

tre(tre(tre(tom, int(4), tom), int(6), tom), int(9),
                                 tre(tom, int(11), tom)) .

tre(tre(tre(tom, int(1), tom), int(3), tom), int(7),
                                  tre(tom, int(2), tom)) .
```

hvor de tre første trærne er sorterte binære trær, mens det siste ikke er det, da vi på høyre side har en node, int(2), med mindre verdi enn rotnode-verdien, noe som ikke er korrekt for binære trær.

Nå som vi har klart deklarasjonen av trær, kan vi lage funksjoner på disse. Vi følger da den samme oppskriften som tidligere for å lage funksjoner. Aktuelle funksjoner for trær og noder kan gå ut på å sette inn nye noder i et tre, fjerne noder fra treet og gjøre et binært søk på treet, hvor vi ser etter et gitt element.

Ettersom treet skal være binært, er funksjonene tilpasset en slik trestruktur. I tillegg skal treet også være sortert. Når vi skal sette inn elementer i treet, blir de undersøkt mot roten av treet, og dermed kan vi enklere søke oss nedover subtrærne mot plassen som passer noden best og sette inn noden der. Variabellisten som er angitt under den som brukes i resten av dette delkapittelet.

```
vars D D' D'' : Data .
vars I I'     : Int .
vars T T' T'' : Tre .
vars U U'     : Tre .
```

```
eq 'put[D tom] = tre(tom, D, tom) .

eq 'put[D (tre(T, D', T'))] = if ((D < D') asBool)
        then tre('put[D T], D', T')
        else tre(T, D', 'put[D T']) fi .
```

Som vi ser kaller vi på funksjonen rekursivt inntil vi finner en ledig plass å sette inn noden. Den første $'put$-funksjonen sørger for at vi har kommet til en bladnode før vi setter inn den nye noden. Vi legger også merke til at vi må bruke *asBool*-funksjonen for å oversette den likhetslogiske funksjonen $D < D'$ til å kunne bruke Maude-maskinens innebygde if-setning. Vi må gjøre dette fordi vi har overskrevet de likhetslogiske funksjonene til å fungere for CMC slik vi ønsker. I CMC forventer vi å få i retur bool(true) eller bool(false), mens i interpreten, som er definert i Maude, får vi i de tilfellene vi bruker Maude sine predefinerte funksjoner true eller false, som beskrevet tidligere (kapittel 6.1).

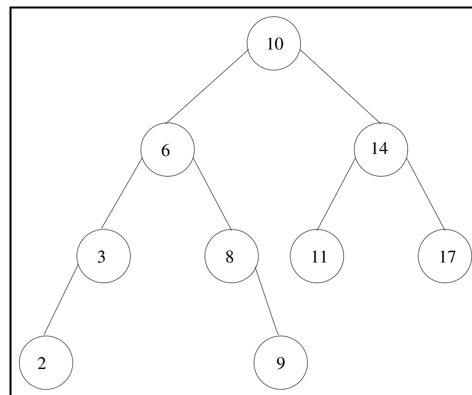
Hvis vi vil sette inn et element i listen, går vi frem som vist ved følgende kjøring av eksempelet i interpreten:

```
>red 'put[int(8) (tre(tom, int(3),
                  tre(tom, int(9), tom)))] .
reduce in INTERPRET : 'put[int(8) tre(tom, int(3),
                            tre(tom, int(9), tom))] .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Tre: tre(tom, int(3), tre(tre(tom, int(8), tom),
                                        int(9), tom))
```

Vi får det ønskede resultatet med $int(8)$ på riktig plass. Elementet har funnet sin plass på høyre side av roten, ettersom den har større verdi enn $int(3)$ og igjen på venstre side av $int(9)$ ettersom den er av mindre verdi her.

I resten av denne seksjonen skal vi vise noen vanlige funksjoner på trær for å belyse de mekanismene i CMC som vi har valgt.

Nå som vi kan sette inn elemeter, skal vi også kunne fjerne elementer. Det som er vanskelig med å fjerne et element fra et tre, er hvordan man skal bygge det opp igjen. Om vi bare skal fjerne en bladnode er saken grei. Om det bare er en rotnode med tomme subtrær er saken også grei. Det som derimot gjør saken vanskeligere er om vi skal fjerne en rot til et tre med subtrær. Det vi må gjøre er å følge regelen [29, 30] ved fjerning av en rotnode. Den sier at vi må først gå ett hakk ned til venstre for roten, og deretter helt ned til bladnoden som er lengst til høyre. På den



Figur 6.3: Tre med rotnode 10

måten får vi tak i det største elementet i det venstre subtreet, samtidig som vi da også opprettholder et sortert binært tre. Vi er nå sikre på at ingen på venstre subtree av den nye roten er større enn selve roten, og at den nye roten er mindre enn alle på høyre subtree. Figurene 6.3 og 6.4 illustrerer hva som skjer når vi fjerner roten med verdien int(10).

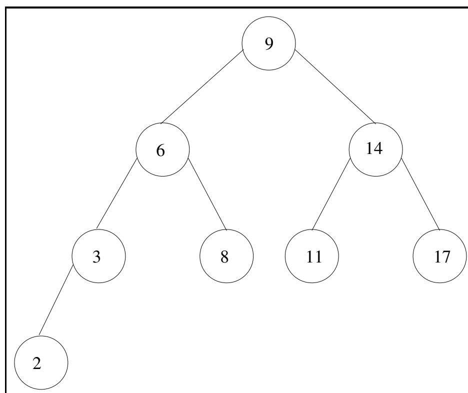
Om vi hadde valgt å ta nærmeste node til venstre, ville det gått ille med tanke på strukturen. Da ville vi ikke like godt være sikre på at treet er et binært tre, da det på venstre side kan forekomme noder med høyere verdi enn den nye roten. Det samme hadde skjedd om vi hadde tatt nærmeste node til høyre. Da ville vi ikke være sikre på at alle noder på høyre subtree ville være større eller lik rotnoden.

På grunn av alt dette, må vi ha en hjelpefunksjon som bygger opp treet igjen, etter at vi har fjernet en node fra treet. Ombyggingen, *'rebuild*, er definert slik, her ved tre likninger da de andre ikke er så ulike de som vises her:

```
eq 'rebuild[tom T] = T .

eq 'rebuild[(tre(T, D, tom)) T'] = tre(T, D, T') .

eq 'rebuild[(tre(T, D, tre(U, D', U'))) T'] =
          tre(T, D, 'rebuild[tre(U, D', U') T']) .
```

Nå er vi klare til å lage funksjonen som fjerner en node, *'remove*, her representert med det to likningene som trengs for å definere den:

```
eq 'remove[D tom] = tom .
```



Figur 6.4: Tre med hvor rotnode 10 er erstattet av 9

```
eq 'remove[D (tre(T, D', T'))] = if (D == D') then
          'rebuild[T T'] else if (D < D')asBool then
            tre(('remove[D T]), D', T')
          else tre(T, D', ('remove[D T'])) fi fi .
```

Vi ser at vi bygger opp et nytt tre når vi fjerner rotnoden ved hjelp av hjelpefunksjonen *'rebuild*. På den måten slås de to barne-trærne sammen til et tre. Når et av barne-trærne er tomt blir det enkelt å fjerne roten.

```
>red 'remove[int(7) tre((tre(tre(tom, int(4), tom),
         int(5), tom)), int(7), tre(tom, int(9), tom))] .
reduce in INTERPRET : 'remove[int(7)
                   tre(tre(tre(tom, int(4), tom), int(5),
    tom), int(7), tre(tom, int(9), tom))] .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Tre: tre(tre(tom, int(4), tom), int(5),
                                  tre(tom, int(9), tom))
```

Vi ser at rotnoden $int(7)$ ble fjernet og vi sitter igjen med med $int(5)$ som den nye rotnoden, da dette var den største verdien på venstre side.

Nå som vi har laget viktige funksjoner for å bygge et binært søketre, kan vi lage funksjoner som undersøker treet. Slike funksjoner kan være å undersøke om treet er et søketre, finne høyden, og gjøre et binært søk, hvor vi ser etter et gitt element. For å vite at treet er et binært søketre, er det viktig å undersøke at også subtrærne er binære søketrær. I et slikt søk, undersøker man rotnoden mot nodene i barne trærne, hvor barnet på høyre side skal være større enn eller lik rotnoden, mens barnet på venstre side skal være mindre enn rotnoden. Her er tre av funksjonene til *'sorted*, som sjekker nodene mot hverandre:

```
eq 'sorted[tom] = bool(true) .

eq 'sorted[tre(tom, D, tom)] = bool(true) .

eq 'sorted[tre(tre(T, D', T'), D, tom)] =
        (D > D') and ('sorted[tre(T, D', T')]) .

eq 'sorted[tre(tom, D, tre(T, D', T'))] =
        (D <= D') and ('sorted[tre(T, D', T')]) .

eq 'sorted[tre(tre(U, D', U'), D, tre(T, D'', T'))] =
        ((D > D') and (D <= D'')) and
             (('sorted[tre(U, D', U')]) and
                   ('sorted[tre(T, D'', T')])) .
```

Her er svaret enten bool(true) om treet er et binært søketre, eller bool(false) om det ikke er det. Ettersom vi ikke bruker metoder der vi forventer å få true eller false som retur, kan vi bruke *and*, som er overskrevet i CMC, og metodene *'less* og *'lessEq* som fungerer på alle datatyper. På den måten sjekker CMC sine verdier bool(true) eller bool(false) mot hverandre. Ettersom alt må stemme overens med hverandre for at vi skal kunne si at det er et binært søketre, er and-funksjonen nyttig å bruke, da kun en bool(false) retur trengs for å avkrefte et søketre-egenskaper.

For å undersøke at et tre er sortert, kan vi bruke de to siste tre-eksemplene over for å vise kjøringer:

```
>red 'sorted[tre(tre(tre(tom, int(4), tom), int(6), tom),
                int(9), tre(tom, int(11), tom))] .
reduce in INTERPRET : 'sorted[tre(tre(tre(tom, int(4), tom),
            int(6), tom), int(9), tre(tom, int(11), tom))] .
rewrites: 21 in 0ms cpu (0ms real) (~ rewrites/second)
result Data: bool(true)


>red 'sorted[tre(tre(tre(tom, int(1), tom), int(3), tom),
          int(7), tre(tom, int(2), tom))] .
reduce in INTERPRET : 'sorted[tre(tre(tre(tom, int(1), tom),
          int(3), tom), int(7), tre(tom, int(2), tom))] .
rewrites: 21 in 0ms cpu (0ms real) (~ rewrites/second)
result Data: bool(false)
```

Vi ser at resultatene blir som ønsket. Det andre treet er ikke et binært søketre da det har node med en mindre verdi enn rotnoden på høyre side.

For å finne høyden på et tre er det viktig å finne den lengste veien fra roten til det bladet som ”henger” lengst ned. Til det trenger vi operasjonen 'max, som sjekker det største av to tall:

```
eq 'max[int(I) int(I')] =
              if (I >= I') then int(I) else int(I') fi .
```

Ved å ha denne hjelpemetoden, kan vi nå lage metoden som beregner den lengste høyden, 'height:

```
eq 'height[tom] = int(0) .
eq 'height[tre(T, D, T')] =
        int(1) + ('max[('height[(T)]) ('height[(T')])]) .
```

En annen viktig funksjon er å kunne søke etter et gitt element i treet.

```
eq 'search[D tom] = bool(false) .
eq 'search[D (tre(T, D', T'))] =
      if ('equal[D D'])asBool then bool(true)
        else if ('less[D D'])asBool then ('search[D T])
            else ('search[D T']) fi fi .
```

Vi ser at denne funksjonen forutsetter at treet er sortert og dermed er dette et binært søk. Om vi hadde ønsket å søke hele treet, kunne vi fjernet if-setningen og erstattet den med funksjonen *or* mellom de tre rekursive kallene.

Denne tre-konstruksjonen er et eksempel på en brukerdefinert datatype. Mulighetene er mange for hva slags typer vi ønsker å ha med i Creol. For at disse skal fungere som datafiler som en del av interpreten, må de importere modulen `EXPR` fra default-datafilen, slik at de får de samme konstruksjonene med hensyn på funksjoner (`Call`-syntaksen) og mulighet til å bruke datatypene der. Dessuten må vi huske å sette dem inn i interpreten, og importere den i modulen `INTERPRET` for at de skal være gyldige i interpreten.

Evalueringen av brukerdefinerte datatyper, ønsket vi også skulle skje automatisk, men som vi så i kapittel 5.1, var det ikke mulig å gjøre om datatypene slik at vi kunne fjerne dem fra interpreten helt med tanke på evalueringer av datatypene. Derfor må det for nye datatyper lages evalueringsfunksjoner, hver gang en brukerdefinert datatype blir innført. Evalueringsfunksjonen for typen `tre` blir som følger:

```
var X X' X''  : Expr .

q eval(tre(X, X', X''), L) =
        tre(eval(X,L), eval(X',L) ,eval(X'',L)) .
```

hvor vi legger merke til at vi kan bruke uttrykk av `Expr` i rotnoden.

Brukerdefinerte subtyper behandles på samme måte som i kapittel 6.5 og vi går ikke nærmere inn på det her.

Kapittel 7

Eksempler på bruk i CMC

I dette kapittelet skal vi se på eksempler som viser Creol brukt i litt større programmer. For hvert eksempel oversetter vi til CMC og ser nærmere på den nye programkonvensjonen og eksekvering i Maude-maskinen.

7.1 Alternating bit protocol

Alternating bit protokollen (ab-protokollen) er et velkjent kommunikasjonsproblem. Det går ut på å sende meldinger fra en avsender til en mottaker, som i sin tur sender en melding tilbake (acknowledgment) for å bekrefte å ha mottatt avsenders melding. Underveis kan meldinger gå tapt og dupliseres, slik at meldingsrekkefølgen kan forandre seg, eller meldingssekvensen kan være inkonsistens.

7.1.1 Grensesnitt

For at sender og mottaker skal kunne kommunisere med hverandre opprettes en kanal. Kanalen fungerer som et lager, som tar vare på meldingene som skal sendes fra sender til mottaker og omvendt. Kanalen er laget slik at den meldingen som kommer inn først, er den som sendes først, slik vi da får en FIFO-kø (first in - first out). Dermed bevarer vi rekkefølgen, som i en liste hvor man setter inn nye elementer bakerst og tar ut elementer fra begynnelsen.

```
interface Channel
begin
    with any
        op put(in x: Data)
        op get(out x: Data)
        op isEmpty(out x: Bool)
end
```

Vi legger merke til at grensesnittet *Channel* tilbyr de tre metoden `put`, `get` og `isEmpty` til omgivelsene. `put` fungerer slik at den tar en verdiparameter som settes in i kanalen, `get` returnerer et element fra kanalen, mens `isEmpty` sjekker om kanalen er tom og returnerer en boolsk verdi.

```
interface Sender
begin
end

interface Receiver
begin
end
```

Som vi ser, tilbyr ikke grensesnittene for *Sender* og *Receiver* metoder til omgivelsene og disse er dermed tomme. Man kunne tenkt seg at de burde tilby metoder for å sende *Sender* meldinger og hente meldinger ut av *Receiver*. Allikevel er disse grensesnittene med fordi de brukes til typingsformål, da all typing er gjort ved grensesnitt. Vi har ingen restriksjoner på disse grensesnittene og kunne brukt supergrensesnittet **any**.

7.1.2 Klasser

For å få i gang prosessen, trenger vi klasser som oppretter objekter. Som start har vi klassen **ABSystem** som setter i gang det hele ved å opprette to kanaler, et *Sender*-objekt og et *Receiver*-objekt. De to kanalene fungerer slik at den ene går fra *Sender* til *Receiver* (*sendR*-kanalen), mens den andre går den andre veien (*recS*-kanalen). *sendR* og *recS* er variable av *Channel*-grensesnittet, mens *S* og *R* er variable av henholdsvis *Sender*- og *Receiver*-grensesnittene.

```
class ABSystem(sendList:list)
begin
    var sendR, recS: Channel,
        S: Sender, R: Receiver
op init ==
    sendR := new ChannelImp();
    recS := new ChannelImp();
    S := new Sender(sendR, recS, sendList);
    R := new Receiver(sendR, recS).
end
```

Vi legger merke til at *Sender* har tre parametre; en liste med elementer som skal gis til *Receiver* og de to kanalene som skal til for å kommuni-

sere med *Receiver*. *Receiver* har kun de to kanalene som parametre.

Klassen **ChannelImp** implementere funksjonene deklarert i grensesnittet *Channel*, det vil si *put*, *get* og *isEmpty*. Ettersom kanalene er implementert med en liste-implementasjon kan Creol ha samme type funksjoner som Standard ML. Da tenker vi på *appendRight*, *length*, *head*, *tail* og *isempty*. Disse er tidligere vist (kapittel 6.3, og implementert i interpreten. Klassen **ChannelImp** kan nå bruke metodene som om de var dens egne.

For å modellere upålitelig overføring av meldinger, implementeres metoden *put* slik at den kan sette inn en melding i listen, miste melding eller duplisere meldingen. Disse fungerer ikke-deterministisk ved bruk av [] som skal velge hvilken metode av disse tre som får kjøre.

```
class ChannelImp
    implements Channel
begin
    var Blist: list = nil
    with any
    op put(in x:Data) == if (x /= null) then
        Blist:= appendRight(Blist,x)
            □
            skip
            □
        (Blist:= appendRight(Blist,x) ;
        Blist:= appendRight(Blist,x))
            fi
    op get(out x:Data) == await not(isempty((Blist)))
            x := head(Blist) ;
            Blist := tail(Blist)
    op isEmpty(out x:bool) == x := (isempty(Blist)).
end
```

Ettersom kanalen kan være tom for meldinger, kan ikke *get* returnere noe før meldingslisten inneholder noe. *await* lager et slippunkt i denne metoden slik vi får bedre prosesstyring. Metoden *isempty* kan brukes før et kall på *get*, da den sjekker listen for innhold.

Nå som vi har kanalen klar, kan vi definere klassene for **Sender** og **Receiver**. Klassen **Sender** har metoder for å sende en melding gjentatte ganger inntil den mottar en bekreftelse og dermed kan sende en ny melding.

Attributtet `cnt` er en teller som brukes til å holde orden på hvor mange

meldinger som er sendt. Denne kan kun bli oppdatert om bekreftelesen fra **Receiver**, `num`, er lik denne. Lengden av listen med meldinger lagres i attributtet `x`. Dette skjer kun før noen meldinger har blitt sendt, slik at `x` er en fast verdi gjennom hele programmet. Når `cnt` har blitt like stor som `x`, vet vi at dette er siste melding. Listen `receivedAck` brukes bare til å sjekke om bekreftelsene har kommet fram og i riktig rekkefølge.

```
class Sender(Outsend: Channel, Inrec: Channel,
        chanList: list)
    implements Sender
begin
    var cnt:int = 0
    var message: Data
    var x:int = 0
    var receivedAck:list

    op run == x := length(chanList) ;
        start ;
        while ( cnt <= x)
        do
            !send ; !get ;
        od
    op get == var num:int ;
        if Inrec /= null
        then
            Inrec.get(; num)
            if (num = cnt)
            then
                receivedAck :=
                appendRight(receivedAck, num)
                start ;
            else
                send ;          fi fi
    op send == (!Outsend.put(pair(message, n)))
    op start == if chanList /= null then
            message := head(chanList);
            chanList := tail(chanList);
            cnt := cnt + 1;
        fi
end
```

Vi legger merke til at vi bruker et par (`pair`) når vi setter inn et element i kanalen. Dermed får vi sendt med både meldingen og en teller, n, som brukes til acknowledgment.

Klassen **Receiver** har metode for å motta en melding og sende en bekreftelse til **Sender** på at den har mottatt meldingen. Alt dette gjøres i metoden *receive*.

`Blist` tar vare på meldingene som blir tatt i mot, `message` lagrer midlertidig meldinger fra **Sender** mens `cnt` brukes til å sjekke om riktig melding har kommet fram.

```
class Receiver(Outsend: Channel, Inrec: Channel)
    implements Receiver
begin
    var Blist:list
    var message: Data
    var cnt:int = 1

    op run == while true do receive od
    op receive == Outsend.get(; message) ;
        if (message /= null)
        then
            if message.scd == cnt then
            Blist := appendRight(Blist, message.fst) ;
            !Inrec.put(cnt) ;
            cnt := cnt + 1 ;
            else skip
        fi fi
end
```

7.1.3 Oversettelse til CMC

Creol programmer forsøker man å oversette så direkte som mulig til CMC. I [10] er det definert regler for hvordan man oversetter og hvordan CMC-koden er definert. Ved å følge disse reglene, sammen med de nye definisjonene gjort tidligere i denne oppgaven, får vi CMC kode som kan kjøres gjennom interpreten.

Kallene som setter det hele i gang, finnes i klassen **'ABSystem.** Vi ser at vi oppretter nye kanaler og nye objekter av **'Send** og **Receive** på samme måte som i Creol.

```
< 'ABSystem : Cl | Inh: nil,
Att: ('sendList : null),
     ('sendR : null), ('recS : null),
     ('S : null), ('R : null),
```

```
Mtds: < 'run : Mtdname | Latt: no,
    Code:
      ('sendR := new 'Channel(nil)) ;
      ('recS := new 'Channel(nil)) ;
      ('S := new 'Send('sendR 'recS 'sendList)) ;
      ('R := new 'Receive('sendR 'recS)) ;
      (end (nil)) > ,
Ocnt: 0
>
```

Kanalen er også forsøkt oversatt direkte, men det er visse forskjeller mellom Creol- og CMC-koden. Som vi ser her er de boolske verdiene `true` og `false` i Creol oversatt til `bool(true)` og `bool(false)` i CMC. Mellom metodene er tegnet * brukt for å skille mellom dem.

```
< 'Channel : Cl | Inh: nil,
Att:  ('Blist : list(nil)),
Mtds: < 'put : Mtdname |
        Latt: 'x : null,
        Code: if( 'x /= null) th
           (('Blist := 'appendRight[('Blist) 'x]))
                []
               empty
                []
           (('Blist := 'appendRight[('Blist) 'x]) ;
           ('Blist := 'appendRight[('Blist) 'x]))
       fi ;
      end(nil) > *
      < 'get : Mtdname |
        Latt: 'x : null,
        Code: (await ('not['isempty['Blist]])) ;
              (('x := 'head['Blist]) ;
              ('Blist := 'tail['Blist])) ;
       end('x) > *
      < 'isEmpty : Mtdname |
        Latt: 'x : bool(true),
        Code: ('x := ('isempty['Blist])) ;
       end('x) >,
Ocnt: 0
>
```

Når det opprettes et nytt **'Send**-objekt, sendes det med tre parametre. Disse tar plass i de tre første plassene i attributtlisten, *Att*. De andre attributtene i listen er ment til internt bruk i selve **'Send**-klassen. Kallet på *'put* er gjort uten å vente på svar. Dermed vet vi ikke om meldingen har blitt satt inn i kanalen, men må vente på en bekreftelse fra **'Receive**.

```
< 'Send : Cl | Inh: nil,
Att: ('Outsend : null),
     ('Inrec : null),
     ('chanList : list(nil)),
     ('cnt : int(0)),
     ('message : null),
     ('x : int(0)),
     ('receivedAck : list(nil)),
Mtds: < 'run : Mtdname |
        Latt: no,
        Code: ('x := 'length['chanList]) ;
              ('start(nil ; nil)) ;
              while ('cnt <= 'x)
              do (
                 ('send(nil ; nil)) ;
                 ('get(nil ; nil)) )
              od  ;
      end(nil) > *
      < 'start :  Mtdname |
         Latt: no,
         Code: (if ('chanList /= null) th
                 ('message := 'head['chanList]) ;
                 ('chanList := 'tail['chanList]) ;
                 ('cnt := ('cnt + int(1)) )
                 fi ) ;
      end(nil) > *
      < 'send : Mtdname |
         Latt: no,
         Code: (! 'Outsend . 'put(pair('message, 'cnt))) ;
      end(nil) > *
      < 'get : Mtdname |
         Latt: ('num : null),
         Code: if ('Inrec /= null) th (
               ('Inrec . 'get(nil ; 'num)) ;

                 if ('num = 'cnt) th
                    ('receivedAck :=
                        'appendRight['receivedAck 'num]) ;
                    ('start(nil ; nil))
                 el ('send(nil ; nil))
                  fi
               ) fi ;
      end(nil) > ,
Ocnt: 0
```

```
>
```

Til slutt har vi klassen **'Receive**, som prøver å motta meldinger hele tiden og sender bekreftelse om meldingen har det riktige nummeret. Om meldingen er feil, kastes den.

```
< 'Receive : Cl | Inh: nil,
Att: ('Outsend : null),
     ('Inrec : null),
     ('Blist : list(nil)),
     ('message : null),
     ('cnt : int(1)),
Mtds: < 'run : Mtdname |
        Latt: no,
        Code: while (bool(true))
                do ('receive (nil ; nil))
              od ;
      end(nil) > *
      < 'receive : Mtdname |
        Latt: no,
        Code: ('Outsend . 'get(nil ; ('message) )) ;
              if (('message /= null)) th
                if ('scd['message] = 'cnt) th
                   ('Blist := 'appendRight['Blist ('fst['message])]) ;
                   (! 'Inrec . 'put('cnt)) ;
                   ('cnt := ('cnt + int(1)))
                el empty
             fi fi ;
      end(nil) > ,
Ocnt: 0
>
```

Det er som vi kan se forskjeller i deklarasjonen av funksjoner i Creol og CMC. Navn på funksjoner oversettes til `Qid` verdier, mens parameterlisten som er omsluttet av vanlige parenteser ((...)) i Creol oversettes til å bli omsluttet av klammeparenteser ([...]) i CMC.

Metodekall oversettes nesten direkte. Bortsett fra navnet på metoden som blir en verdi av `Qid`, definisjonen av **in**- og **out**-parametrene som vi kan se likt.

En litt mer vesentlig forskjell mellom Creol og CMC er datatypene som i Creol er angitt som i Java. For eksempel er en integer representert med tallet 1, mens den i CMC oversettes den til å være omsluttet av navnet på datatypen den er, altså int(1). Dette gjelder for alle datatyper. Her må

man ved oversettelse være forsiktig med subtyper, slik at de blir deklarert som subtyper også i CMC.

7.1.4 Eksekvering

Et kall på dette programmet, gjøres ved å opprette et nytt objekt av **'AbSystem**, hvor elementene i listen kan være alle slags datatyper som er definert i Creol:

```
new 'ABSystem (list(str("Her") str("er")
                  str("en") str("melding"))) .
```

Dermed kan vi se resultatet av eksekveringen. Det er viktig ettersom vi vil finne ut om alle meldingene kommer fram og i riktig rekkefølge. Omskrivingen `rew` er litt farlig å bruke da den kan favorisere et objekt og bruke all prosessor kraft på det. Omskrivingen `frew` er mer rettferdige og bytter på objektene. Vi kjører med både `rew` og `frew` og legger merke til at programmet terminerer ved begge forsøk, og med samme svar. Sluttilstanden som er nådd, forteller oss at alle meldingene har kommet fram og i riktig rekkefølge.

Ved å legge en begrensning på antall omskrivinger kan vi se hva som skjer. En omskriving med `frew` i 100 steg, hvor den oversatte CMC-koden for ab-protokollen blir gitt med sammen kallet på programmet, gir oss denne kanalen fra **Sender** til **Receiver**:

```
< 'Channel0 : Ob |
  Cl: 'Channel,
  Pr: empty,
  PrQ: none,
  Lvar: ('caller : 'Receive0),('label : int(3)),
        'x : pair(str("Her"), int(1)),
  Att: ('this : 'Channel0),'Blist : list(nil),
Lcnt: 2 >
```

Vi ser at den første meldingen er lagret som et par i `'x`. Etter 500 omskrivingssteg har alle meldingene kommet fram til **Receiver**.

```
< 'Receive0 : Ob |
  Cl: 'Receive,
  Pr: (15 ?('message)) ;
      if bool(true) th
       if 'equal[('scd['message]) 'cnt]
          th ('Blist :=
           ('appendRight['Blist ('fst['message])])) ;
           ! 'Inrec . 'put('cnt) ;
```

```
            'cnt := ('plus['cnt int(1)])
        el empty
        fi
       el empty
       fi ;
      end(nil) ;
      continue(14),
   PrQ: ((await 1 ? ; 1 ?(nil)),no) :
          (await 14 ? ; (14 ?(nil)) ;
        while bool(true) do
           'receive(nil ; nil)
        od ;
        end(nil) ; continue(1)),
        ('caller : 'Receive0),'label : int(1),
   Lvar: ('caller : 'Receive0),'label : int(14),
   Att: ('this : 'Receive0),('Outsend : 'Channel0),
        ('Inrec : 'Channel1),
        ('Blist : list(str("Her") str("er") str("en")
             str("melding"))),
        ('message : pair(str("melding"), int(4))),
        'cnt : int(5),
Lcnt: 16 >
```

Vi legger merke til at både `Pr` har ventende kode, mens `PrQ` har ventende slippunkter. Dette fordi vi har en while-setning i **Receiver** som alltid er sann. Den siste meldingen blir lagret i `'message` og blir overskrevet bare når det kommer en ny melding.

7.1.5 Søk

Vi så fra kjøringen over at alle meldingene kom fram. Ettersom protokollen er modellert usikkert og skal velge ikke-deterministisk mellom å legge meldingen som vanlig i kanalen, miste meldingen, eller duplisere meldingen og legge to av den i kanalen, ønsker vi å finne ut om det finnes tilstander hvor vi får meldingen i feil rekkefølge. Vi begynner med å søke på en sluttilstand hvor alle meldingene har kommet frem og i riktig rekkefølge til **Receiver**:

```
search [1] init =>* C:Configuration
 < 'Receive0 : Ob | Cl: 'Receive,
 Pr: P:ProgList,
 PrQ: W:MProg,
 Lvar: L:Subst,
 Att: ('this : 'Receive0),('Outsend : 'Channel0),
      ('Inrec : 'Channel1), ('Blist : list(str("Her")
```

```
                 str("er") str("en") str("melding"))),
 IA:InitSubst,
 Lcnt: N:Nat > .
```

hvor init er klassen oversatt til CMC. Vi går tom for minne og får dermed ikke noe svar ut. Fordi programmet er ikke-deterministisk generes mange tilstander, men selv om vi fjerner ikke-determinismen er søket fortsatt for stort.

Vi forsøker å dele opp søket og ser etter en tilstand hvor bare meldingen $str(''Her'')$ har kommet til mottakeren.

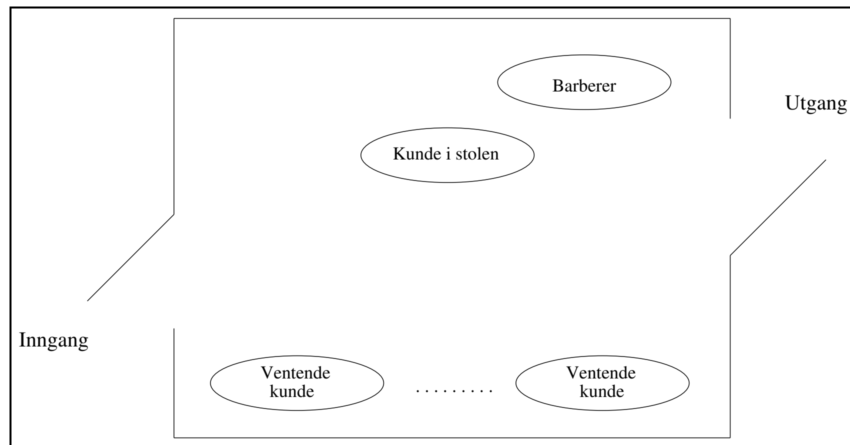
```
search [1] init =>* C:Configuration
< 'Receive0 : Ob | Cl: 'Receive,
  Pr: P:ProgList,
  PrQ: W:MProg,
  Lvar: L:Subst,
  Att: ('this : 'Receive0),
       ('Outsend : 'Channel0),
       ('Inrec : 'Channel1),
       ('Blist : list(str("Her"))), IA:InitSubst,
Lcnt: N:Nat > .
```

Her får vi et svar på at denne tilstanden finnes. Dermed prøver vi oss på en tilstand hvor vi søker kanalen for innehold av to like meldinger.

```
search [1] init =>* C:Configuration
 < 'Channel0 : Ob | Cl: 'Channel,
   Pr: P:ProgList,
   PrQ: W:MProg,
   Lvar: L:Subst,
   Att: ('this : 'Channel0),
        ('Blist : list(pair(str("Her"), int(1))
             pair(str("Her"), int(1)))), IA:InitSubst,
Lcnt: N:Nat > .
```

Også her finner Maude-maskinen en tilstand. Dette kan ha noe å gjøre med at **Receiver** ikke har rukket å sende ut en bekreftelse til **Send** før den *Her* sendes på nytt. Vi prøver å nå et søk på en tilstand hvor meldingen $str(r")$ har blitt borte i kanalen.

```
search [1] init =>* C:Configuration
< 'Channel0 : Ob |
  Cl: 'Channel,
  Pr: P:ProgList,
  PrQ: W:MProg,
```



Figur 7.1: Den sovende barbereren

```
  Lvar: L:Subst,
  Att: ('this : 'Channel0),
       ('Blist : list(pair(str("Her"), int(1))
             pair(str("en"), int(3)))), IA:InitSubst,
Lcnt: N:Nat > .
```

Dette søket ga ingen svar og vi må konkludere med at det blir altfor mange tilstander til at Maude-maskinen kan håndtere et så komplekst program som dette. Det er også en indikasjon på noe positivt at vi ikke får svar her. Vi vet at **Send** ikke sender ut neste melding før den har mottatt bekreftelse på forrige melding, og da kan vi se positivt på at Maude-maskinen ikke klarer å finne noen tilstander som skal oppfylle noe vi ikke ønsker.

7.2 Sovende barberer problemet

Det klassiske "sovende barberer" problemet [8] består av en barberersalong med to dører og noen stoler. To prosesser er viktige i denne sammenhengen; barbereren og kunden. Disse prosessene skal samarbeide for å fullføre oppdraget. Når en kunde vil klippe håret, går han inn den ene døra som leder til salongens venterom. Er venterommet fullt, tar kunden seg en tur, før han kommer tilbake og prøver på nytt. Er det plass i venterommet tar kunden en plass og venter på at turen hans skal komme og at han kan ta plass i barberer-stolen. Er det ingen i venterommet eller barberer-stolen når kunden kommer inn, går kunden og setter seg i barberer-stolen. Figur 7.1 illustrerer salongen med barbereren og kundene.

Hvis barberer-stolen er opptatt med en annen kunde, og det er en ledig plass i venterommet tar kunden plass der. Om det ikke er kunder i venterommet, tar barbereren en sove-pause. Barbereren blir vekket av at en kunde setter seg i barberer-stolen.

Når kunden er ferdig klipt går han ut gjennom den andre døren som eksisterer i salongen. Det er opp til barbereren å slippe ut kunden, ettersom det er barbereren som har kontroll på når kunden er ferdig klipt.

Dette problemet illustrerer relasjonen mellom en klient og en server som ofte eksisterer mellom prosesser. Her er det nødvendig å synkronisere prosessene.

7.2.1 Grensesnitt

I en barberer-salong finnes en barberer og ingen eller flere kunder. Disse har grensesnitt som er definert tomme og tilbyr ikke metoder til omgivelsene:

```
interface Barber
begin
end

interface Customer
begin
end
```

Disse grensesnittene brukes kun til typingsformål, ettersom all typing er gjort ved grensesnitt.

Både kunden og barbereren har egne metoder som de tilbyr. Kunden tilbyr metoden `getHaircut` når håret skal klippes:

```
interface BarberShop
begin
    with Customer
        op getHaircut(out return:bool)
end
```

Denne har en boolsk variabel **return** som er ment å fortelle kunden om det er plass i venterommet eller ikke. Kunden som kommer først inn i salongen plasseres først i køen, og dermed får vi en FIFO-kø. Køen tar bare et maks antall, så når det antallet er oppnådd, må kundene som

finner fulle venterom ta seg en tur og komme tilbake senere.

Barbereren tilbyr to metoder, `getNextCustomer` og `finishedCut` som henholdsvis henter ny kunde og avslutter en jobb:

```
interface WaitingRoom
begin
    with Barber
        op getNextCustomer
        op finishedCut
end
```

For at kunde og barberer skal kunne kommunisere med hverandre, opprettes et grensesnitt som arver `BarberShop` og `WaitingRoom`:

```
interface Work
    inherits BarberShop, WaitingRoom
begin
end
```

Her tilbys metodene som arves fra `BarberShop` og `WaitingRoom` til omgivelsene.

7.2.2 Klasser

Klassen `Barber` er den som setter det hele i gang med opprettelse av salongen og kunder. Kundene får med seg salong-attributtet som parameter.

```
class Barber
    implements Barber
begin
    var shop: WRoom,
    with any
    op run ==
        var c0, c1, c2, c3, c4: Customer
        shop := new Wroom();
        c0 := new Customer(shop);
        c1 := new Customer(shop);
        c2 := new Customer(shop);
        c3 := new Customer(shop);
        c4 := new Customer(shop);
        ! work(nil) ;
```

```
    op work ==
        (shop . getNextCustomer) ;
        (await wait) ;
        (shop . finishedCut) ;
        (! work(nil)) ;
end
```

Klassen **Customer** har et kall for å be å et hårklipp. Er venterommet fullt (attributtet `back` er lik true), prøver kunden igjen etter en liten stund.

```
class Customer(shop: Wroom)
    implements Customer
begin
    with any
    op run ==
        var back:bool ;
        (shop . getHaircut(this ; back)) ;
        if (back) then
            (await wait) ; run
        else skip fi ;
end
```

Vi kunne hatt innebygget **await**, hvor vi kalleren venter på returkall om at det er plass i venterommet, men på den måten venter kunden utenfor inngangen til barber-butikken til det er et sete ledig.

Klassen for salongen, **WRoom** implementerer grensesnittet **Work** og definerer de tre metodene derfra. To av disse, `getNextCustomer` og `finishedCut`, blir kalt av barbereren. I `getNextCustomer` venter barbereren til det sitter en kunde i stolen før han starter å jobbe. Når barbereren er ferdig med kunden signaliserer han at stolen er ledig.

Metoden `finishedCut` blir kalt på når `getNextCustomer` er ferdig. Her åpnes døren slik at kunden kan dra. Om det ikke er flere i køen som venter på sin tur, går barbereren i ventemodus.

Kunden kaller på metoden `getHaircut`. Her finner kunden ut av om det er plass i venterommet. Er det ikke plass, returneres en boolsk variabel som forteller kunden om å komme tilbake senere. Er det plass, venter kunden på sin tur til å sette seg i stolen.

Vi legger merke til at attributtet `barber` blir satt til **true** av den første kunden, og blir satt til **false** igjen av barbereren først når det ikke er flere kallere igjen som er på vent. Attributtet `barberAvail` skifter på å

være **true** og **false**, avhengig av om barbereren klipper en kunde eller ikke.

Listen over kunde som har vært innom salongen er i `queue`. Denne brukes bare for å undersøke om kundene som kalte på en metoden har kommet gjennom. Vi legger merke til at vi bruker attributtet `caller` som er innbygget og forteller hvem som er kalleren. På den måten trenger vi ikke sende med `this` (som inneholder navnet på kalleren) fra kalleren.

Attributtet `chair` sier ifra om stolen er besatt av en kunde eller ikke. Når den er ledig kan en ny kunde sette seg for en hårklipp.

Når en kunde er ferdig klipt åpner barbereren døren ved å sette `door` til **true**, og dermed kan kunden dra ved å sette `customerLeft` til **true**.

```
class WRoom
    implements Work
begin
    var barber:bool = false
    var barberAvail:bool = true
    var queue:list
    var chair:bool = true
    var customerLeft:bool = false
    var door:bool = false
    var begrens:int = 0
    var done:int = 0

with Customer
    op getHaircut(out return:bool) ==
        if (begrens <= 2)           then
            begrens := begrens + 1 ;
            barber := true ;
            await (barberAvail) ;
            barberAvail := false ;
            queue := queue + list(caller) ;
            await (chair) ;
            chair := false ;

            await (door) ;
            door := false ;
            customerLeft := true ;
            return := false ;
            begrens := begrens - 1
        else
            return := true
```

```
        fi

with Barber
    op getNextCustomer ==
        await (barber) ;
        barberAvail := true;
        await (not(chair)) ;
        chair := true ;

    op finishedCut ==
        door := true ;
        await not(door) ;
        await customerLeft ;
        customerLeft := false ;
        door := false ;
        done := done + 1 ;
        if (begrens < 1)
        then
            barber := false ;
        fi
end
```

I eksempelet over har vi satt en grense på maks 2 i venterommet. Attributtet `done` forteller hvor mange frisøren har klipt. Denne er alltid større enn eller lik antall kunder som har vært i stolen. Antall kunder som har vært i stolen er i sin tur større enn eller lik antall kunder som har sittet i venterommet.

7.2.3 Oversettelse til CMC

Ved å bruke oversettingsreglene, får vi disse klassene for **Barber** og **Customer**:

```
< 'Barber : Cl | Inh: nil,
Att: ('shop : null),
Mtds: < 'run : Mtdname |
      Latt: ('c0 : null), ('c1 : null), ('c2 : null),
            ('c3 : null), ('c4 : null),
      Code: ('shop := new 'WRoom(nil)) ;
            ('c0 := new 'Customer('shop)) ;
            ('c1 := new 'Customer('shop)) ;
            ('c2 := new 'Customer('shop)) ;
            ('c3 := new 'Customer('shop)) ;
            ('c4 := new 'Customer('shop)) ;
```

```
            (! 'work(nil)) ;
          (end(nil)) > *
      < 'work : Mtdname |
      Latt: no,
      Code: ('shop . 'getNextCustomer(nil ; nil)) ;
            (await wait) ;
            ('shop . 'finishedCut(nil ; nil)) ;
            (! 'work(nil)) ;
          end(nil) >,
Ocnt: 0
>

< 'Customer : Cl | Inh: nil,
Att: ('shop : null), ('back : null),
Mtds: < 'run : Mtdname |
      Latt: no,
      Code:
        ('shop . 'getHaircut(nil ; 'back)) ;
        if ('back) th
           (await wait) ;
           'run(nil ; nil)
        el empty fi ;
       end(nil) >,
Ocnt: 0
>
```

Klassen **Wroom** representeres slik:

```
< 'WRoom : Cl | Inh: nil,
Att: ('barber : bool(false)), ('barberAvail : bool(true)),
     ('queue : list(nil)), ('chair : bool(true)),
     ('customerLeft : bool(false)), ('door : bool(false)),
     ('begrens : int(0)), ('done : int(0)),
Mtds: < 'getHaircut : Mtdname |
        Latt:  ('return : null),
        Code:
              if ('begrens <= int(2)) th
                 ('begrens := 'begrens + int(1)) ;

                 ('barber := bool(true)) ;

                 (await ('barberAvail)) ;
                 ('barberAvail := bool(false)) ;

                 ('queue := 'queue + list('caller)) ;
```

```
                 (await ('chair)) ;
                 ('chair := bool(false)) ;

                 (await ('door)) ;
                 ('door := bool(false)) ;

                 ('customerLeft := bool(true)) ;
                 ('return := bool(false)) ;
                 ('begrens := 'begrens - int(1))
              el ('return := bool(true))
              fi ;
      end('return) > *
      < 'getNextCustomer : Mtdname |
        Latt: no,
        Code: (await ('barber)) ;

              ('barberAvail := bool(true)) ;
              (await ('not['chair])) ;


              ('chair := bool(true)) ;
      end(nil) > *
      < 'finishedCut : Mtdname |
        Latt: no,
        Code: ('door := bool(true)) ;
              (await ('not['door])) ;

              (await ('customerLeft)) ;
              ('customerLeft := bool(false)) ;

              ('door := bool(false)) ;
              ('done := 'done + int(1)) ;

              if ('begrens < int(1))
                th ('barber := bool(false)) fi ;
      end(nil) >,
Ocnt: 0
>
```

7.2.4 Eksekvering

Ved eksekvering av dette eksempelet ønsker vi å undersøke om alle kundene har fått seg en hårklipp. Rekkefølgen på kundene er ikke viktig. Det

viktigste er at alle prosessene klarer å samarbeide og fullføre oppdraget.

Vi har laget et attributt, `begrens`, som skal holde orden på antall plasser i venterommet. Attributtet `queue` lagrer alle kunder som har kommet seg til venterommet, mens attributtet `done` teller hvor mange kunder barbereren ha frisert.

Ved å kjøre med omskrivingen `rew`, velges en prosess som kjører uten å slippe til de andre. Dermed prøver vi oss med `frew` uten å angi antall steg. Programmet terminerer og vi får dette i **WRoom**:

```
< 'WRoom0 : Ob |
  Cl: 'WRoom,
  Pr: empty,
  PrQ: (await 'barber ;
       ('barberAvail := bool(true)) ;
       await ('not['chair]) ;
       ('chair := bool(true)) ;
       end(nil)),
       ('caller : 'Barber0),
       'label : int(18),
  Lvar: ('caller : 'Barber0),
        'label : int(16),
  Att: ('this : 'WRoom0),
       ('barber : bool(false)),
       ('barberAvail : bool(true)),
       ('queue : list('Customer0 'Customer1
             'Customer2 'Customer3 'Customer4)),
       ('chair : bool(true)),
       ('customerLeft : bool(false)),
       ('door : bool(false)),
       ('begrens : int(0)),
       'done : int(5),
Lcnt: 2 >
```

Vi legger merke til at alle kundene er i køen, `'queue`, i den rekkefølgen de ble opprettet. Dessuten angir `'done` antall kunder barbert, noe som stemmer med int(5) og `'begrens` er int(0), som forteller oss at venterommet er tomt.

Barbereren venter fortsatt på flere kunder, derfor er det fortsatt kode i `PrQ`. `'barber` venter på at bli sann igjen, noe som skjer først når en kunde kommer inn i salongen. Bortsett fra `'done` og `'queue`, som brukes til å undersøke spesifikasjonen, har alle attributtene initialtilstand igjen, noe som er bra med tanke på at salongen er tom igjen.

7.2.5 Søk

Fra kjøringen over så vi at alle kundene fikk vært hos barbereren. Vi ønsker å søke for å finne ut om det finnes en sluttilstand hvor ikke alle kundene har fått være innom. Vi bruker telleren `'done` til å undersøke dette:

```
search [1] in Barber : init =>! C:Configuration
< 'WRoom0 : Ob |
  Cl: 'WRoom,
  Pr: P:ProgList,
  PrQ: W:MProg,
  Lvar: L:Subst,
  Att: IA:Subst,
       'done : int(I:Int),
Lcnt: N:Nat >
          such that I:Int <= 4 = true .
```

Vi får ikke noe resultat, akkurat som med store søk i AB-protokollen. I dette tilfellet er det positivt da vi ikke ønsker at denne tilstanden skal inntreffe.

Vi kan se på en enkel tilstand hvor attributtet `'queue` består av den første kunden som kalte på *'getHaircut*:

```
search [1] init =>* C:Configuration
< 'WRoom0 : Ob |
  Cl: 'WRoom,
  Pr: P:ProgList,
  PrQ: W:MProg,
  Lvar: L:Subst,
  Att: IA:Subst,
       ('queue : list('Customer0)),
       IA':Subst,
Lcnt: N:Nat > .
```

Vi får det resultatet vi er ute etter, da dette tilfellet finnes

Ved å undersøke tellerne `'begrens` og `'done`, kan vi se om vi har kommet til en tilstand hvor `'begrens` er større enn `'done`:

```
search [1] in Barber : init =>* C:Configuration
< 'WRoom0 : Ob |
  Cl: 'WRoom,
  Pr: P:ProgList,
  PrQ: W:MProg,
```

```
  Lvar: L:Subst,
  Att: IA:Subst,
       ('begrens : int(I:Int)),
       'done : int(I':Int),
Lcnt: N:Nat >
      such that I:Int > I':Int = true .
```

Her får vi ut et ønsket svar. Det finnes en tilstand hvor betingelsen stemmer. Her er `'begrens` lik 1 og `'done` lik 0. Dette betyr at det er en kunde i salongen som ikke er ferdig klipt enda. Ved å søke etter alle slike tilstander får vi flere svar noe som er positivt da det skal stemme inntil antall ferdig klipte er flere enn det kan være i venterommet.

Kapittel 8

Konklusjon

I denne oppgaven har vi sett på innføringen av brukerdefinerte datatyper og funksjoner i Creol, og hvordan de er definert i CMC. Interpreten ble også tilpasset datatypene og funksjonene. Målet med oppgaven var å finne en enkel og elegant løsning for denne innføringen. Vi har sett at vi nå enkelt kan slippe å oppdatere interpreten for nye funksjoner. Når det gjelder brukerdefinerte datatyper, som er definert ved konstruktører og som er uavhengige fra interpreten, må vi derimot oppdatere interpreten ved at vi trenger nye evaluerings-likninger ved nye datatyper (se kapittel 8.1).

Dermed oppnådde vi ikke fullstendig uavhengighet. Grunnen til det er evalueringen av datatypene som foregår i interpreten. Vi gjorde forsøk på å frigjøre datatypene fra interpreten på samme måte som funksjonene, men resultatet var at Maude-maskinen ikke forstod at datatypene var datatyper, men trodde at de var funksjoner.

Ved å undersøke problemstillingene fra kapittel 1.1 kan vi prøve å oppsummere det som er oppnådd i denne oppgaven. Eksemplene som er brukt i denne oppgaven, interpreten og datafilene finnes i [6].

8.1 Problemstillingene

Vi begynner med å se på delproblemstillingene.

Delproblemstilling 1

Den første delproblemstillingen lød slik:

- Hvordan kan CMC utvides med brukerdefinerte datatyper og funksjoner på en enkel og naturlig måte?

Vi har sett på innføringen av nye datatyper og funksjoner. Funksjoner oversettes automatisk fra Creol til CMC. Evalueringen fungerer for alle funksjoner som er deklarert slik det er beskrevet i kapittel 6.1. Datatypene oversettes også direkte, om de er predefinerte i Creol. Om de derimot er brukerdefinerte må de lages en egen evalueringsfunksjon for disse som settes inn i interpreten.

For eksempel når vi innfører trær (kapittel 6.6), får vi en ny sort `Tree` som er en subsort av `Data` og en ny konstruktør:

```
op tre  : Tre Data Tre -> Tre [ctor] .
```

Disse kan ligge på en fil separat fra interpreren, men vi trenger også en likning for å evaluere tree. Denne må ligge i interpreten og er definert slik:

```
eq eval(tre(X, X', X''), L) =
        tre(eval(X,L), eval(X',L) ,eval(X'',L)) .
```

hvor X, X' og X'' er uttrykk av `Expr` og L er en `Subst`-liste.

Delproblemstilling 2

Delproblemstilling 2 lyder som følger:

- Hvordan fungerer Creol/CMC når man utvider anvendelsesområdet til å omfatte et større spekter av datatyper og anvendelser på både de nye typene og de som er definert fra før?

Det har blitt enklere å definere nye funksjoner. Definisjonen av disse berører ikke interpreten i den forstand de gjorde det tidligere, da vi ikke trenger å definere egne evalueringslikninger for funksjoner. Dette har gjort interpreten enklere i den forstand at vi nå har mindre kode for å utføre samme oppdrag som tidligere.

Delproblemstilling 3

Til slutt har vi den siste delproblemstillingen:

- Hvordan vil innføring av subtyper, både predefinerte og brukerdefinerte kunne gjennomføres i interpreten?

Ved å introdusere subtyper av datatyper, håpet vi på at disse kunne bruke supertypene sine funksjoner. Dette viste seg mulig, men også farlig da supertypene har et større anvendelsesområdet enn subtypene. På den måten kan ulovlige hendelser inntreffe ved subtype-bruk (kapittel 6.5. Det positive med subtyper, var at vi kunne definere funksjoner for disse, hvor vi ikke fikk gå utover anvendelsesområdet takket være typeanalysen i Maude.

Hovedproblemstilling

Nå kan vi prøve å svare på hovedproblemstillingen som lyder slik:

- Hvordan vil utvidelser i Creol påvirke den abstrakte maskinen, spesielt med tanke på innføring av brukerdefinerte datatyper og funksjoner?

Den abstrakte maskinen måtte modifiseres til å takle den nye definisjonen på funksjoner, som ble definert ved hjelp av **Call**-syntaksen 6.1. Typesjekkingen måtte modifiseres til å takle denne syntaksen. Dette førte til at CMC-programmer med gammel syntaks ikke lenger kunne interpreteres gjennom den nye typesjekkingen. Gamle Creol-programmer derimot, måtte oversettes på nytt, slik at de ble tilpasset de nye funksjonsdeklarasjonene. Dette hadde ingen innvirkning på resten av interpreten.

På grunn av **Call**-syntaksen ble da definisjonen av funksjoner i CMC forandret. Dette førte til justeringer i CMC, da funksjoner nå blir definert annerledes. Datatypene beholder den samme definisjonen.

Overlasting av funksjoner er nå lov i henhold til **Call**-syntaksen og evalueringslikningen som utføres på denne. Dessuten ble evalueringsfunksjoner for datatyper som kunne overlastes ulovlig erstattet med mer sikre evalueringer 6.2.

8.2 Videre arbeid

Det langsiktige målet for Creol-prosjektet er å studere åpenhet i distribuerte objektorienterte systemer. Det finnes flere retninger i Creol man kan jobbe videre med for å lage et mest mulig robust system. Noen av forslagene nedenfor kan kanskje implementeres i Creol og språkets verktøy.

Denne oppgaven har sett på implementasjonen av funksjoner og datatyper i sammenheng med CMC og interpreten. Vi har ikke gått inn på den direkte oversettelsen fra Creol til CMC. Et område å se på her er hvordan man skal skille definisjoner av datatype fra definisjoner av funksjoner, som på dette tidspunktet er like hverandre.

Det finnes mange datatyper man kunne tenke seg å innføre. Vi så på en eksplisitt datatype i SML, men det finnes også et sett av typer man også kunne tenke seg å innføre. Et eksempel er ukedager, som i SML kan defineres i et sett. Hvordan vil dette kunne løses i Maude? Ved hjelp av konstruktører for hver eneste dag, eller på en annen måte? Et annet område er subtyper av brukerdefinerte datatyper. For eksempel subtyper av

typen tre kan være trær av kun naturlige tall eller et tomtre. Det ble ikke gitt en løsning på det her, noe som kunne vært en ide å ha fått til.

En ide er å tilby unntakshåndtering i interpreten. En slik håndtering kan slå til om det man kaller noe som ikke gir svar fordi et annet objekt henger. Ved metodekallet **o.m**, hvor **o** er objektet og **m** er metodenavnet, kan det hende at **o** er borte eller **o** er lik null. Da kan enten programmet avsluttes ved bruk av unntakshåndtering, eller en timeout-mekanisme implementeres slik at objektet som henger slipper taket.

Det jobbes med å utvikle socket support i fremtidige versjoner av Maude. Dette kan i fremtiden utnyttes i Creol, da man kan ha kommunikasjon med for eksempel filer, hvor man kan lese fra og skrive til filer.

Bibliografi

[1] Creol nettside. http://www.ifi.uio.no/~creol/.

[2] Maude nettside. http://maude.cs.uiuc.edu/.

[3] prelude.maude. http://maude.cs.uiuc.edu/versions/2.1/prelude.maude/.

[4] Sri international. http://www.sri.com/.

[5] Sun's javadoc. http://java.sun.com/j2se/1.4.2/docs/api/.

[6] Interpret, datafiler og eksempler. http://heim.ifi.uio.no/~nabilms/eksempler/.

[7] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[8] Gregory R. Andrews. *Multithreaded, Parallel, and Distributet Programming.* Addison-Wesley, 2000.

[9] Einar Broch Johnsen, Olaf Owe and Marte Arnestad. Combining Active and Reactive Behavior in Concurrent Objects. *In. Proc. of the Norwegian Informatics Conference (NIK'03)*, 2003. Tapir.

[10] Marte Arnestad. En abstrakt maskin for Creol i Maude. Hovedfagsoppgave, University of Oslo, 2003.

[11] Eyvind Wærsted Axelsen. A Meta-Level Framework for Recording and Utilizing Communication Histories in Maude. Hovedfagsoppgave, University of Oslo, 2004.

[12] R. Ben-Natan. *CORBA: A Guide to Common Object Request Broker Architecture.* McGraw-Hill, New York, 1995.

[13] David Flanagan. *Java in a Nutshell.* O'Reilly, 4th Edition, 2002.

[14] Stephen Gilmore. *Programming in Standard ML '97: An On-line Tutorial.* Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.

[15] Rachel Harrison. *Abstract data types in standard ML.* Wiley, 1993.

[16] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd Edition, 1998.

[17] Bjørn Kristoffersen. *Funksjonell programmering i standard ML*. Department of Informatics, University of Oslo, 1994. kompendium 61.

[18] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 1992.

[19] O.J. Dahl, K. Nygaard and B. Myrhaug. *Simula 67 Common Base Language*. Technical Report S-2, Norwegian Computer Center, Oslo, 1968.

[20] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributes Concurrent Objects. *Technical report*, 2004.

[21] Einar Broch Johnsen and Olaf Owe. Inheritance in the Presence of Asynchronous Method Calls. *Department of Informatics, University of Oslo*, 2004.

[22] Marcus Persson. Kompilator fra OUN til Java. Hovedfagsoppgave, University of Oslo, 2002.

[23] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2002.

[24] Olaf Owe and Isabell Ryl. On combining object orientation, openness and reliability. *In. Proc. of the Norwgian Informatics Conference (NIK'99)*, 1999.

[25] Olaf Owe and Isabell Ryl. On Combining Reasoning Control with Distribution and Openness in Object Oriented Systems. *Research report*, 1999.

[26] Olaf Owe and Isabell Ryl. Oun a formalism for open, object oriented distributed systems. *Research report 270*, 2003. Tapir.

[27] Olaf Owe and Isabell Ryl. Oun Grammar. *Technical report*, 2003.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and Carolyn Talcott. Maude Manual (Version 2.1), 2004.

[29] Ulf Uttersrud. *Algoritmer og datastrukturer - Kompendium*. Høgskolen i Oslo, 2004.

[30] Mark Allen Weiss. *Data Structures* & *Algorithm Analysis in Java*. Addison-Wesley, 1998.

[31] Akinori Yonezawa. ABCL: An Objekt-Oriented Concurrent System. *Series in Computer Systems*, 1990.

[32] Peter Csaba Ølveczky. *Formal Modeling and Analysis of Distributed Systems in Maude*. Compendium INF3230/INF4230, Department of Informatics, University of Oslo, 2003.