TEKNILLINEN KORKEAKOULU
Teknillisen fysiikan koulutusohjelma

ERIKOISTYÖ
Tik-61.195 Informaatiotekniikka
14.4.1992

Oliojärjestelmän laajentaminen metaobjektiprotokollan avulla

Ora Lassila
29962w, Tf N

Oliojärjestelmän laajentaminen metaobjektiprotokollan avulla

Ora Lassila

Sisällysluettelo

1. Työn taustaa 3
1.1. CLOS 3
1.2. Metaobjektiprotokollat 4
1.3. BEEF 4
2. CLOSin metaobjektiprotokolla 7
2.1. Aliprotokollat 7
2.1.1. Metaolioiden alustusprotokolla 8
2.1.2. Luokkien viimeistelyprotokolla 9
2.1.3. Ilmentymien rakenneprotokolla 10
2.2. Toteutuksia 10
3. Macintosh Common Lisp ja sen MOP 11
3.1. Olemassaolevat piirteet 11
3.2. Metaobjektiprokollan puuttuvien piirteiden lisäys MCL:iin 12
3.2.1. Metaolioiden alustusprotokolla 12
3.2.2. Luokkien viimeistelyprotokolla 13
3.2.3. Ilmentymien rakenneprotokolla 14
4. BEEFin muuttaminen CLOS-ohjelmaksi 16
4.1. Luokkien määrittely 16
4.2. Lovien käsittely 16
4.2.1. Maailmaherkät lovet 16
4.2.2. Dynaaminen perintä 17
4.2.3. Automaattiset käänteislovet 17
4.2.4. Demonit 17
4.3. Metodit 19
5. Yhteenveto 20
6. Lähdeviitteet 21
7. Ohjelmalistaukset 22

1. Työn taustaa

Tämä työ koostuu kolmesta eri osasta, joilla pyritään ymmärtämään Common Lisp Object Systemin (CLOS) metaobjektiprotokollan olemusta:

- Esitetään yhteenveto ja luonnehdinta Common Lisp Object Systemin metaobjektiprotokollasta.
- Laajennetaan eräs Common Lisp -kääntäjä toteuttamaan osia metaobjektiprotokollasta.
- Esitetään yksityiskohtainen suunnitelma Common Lispillä kirjoitetun kehysjärjestelmän sovittamiseksi CLOS-yhteensopivaksi metaobjektiprotokollan avulla.

1.1. CLOS

CLOS eli Common Lisp Object System [Steele 90] on tulevan ANSI X3J13 -standardin (American National Standard for Common Lisp) osa. CLOS on Common Lispin laajennus, ja sen tarkoituksena on tehdä Common Lispistä oliokeskeinen ohjelmointikieli. CLOSissa on neljä merkittävää osaa tai piirrettä:

- Geneeriset funktiot,
- Moniperintä,
- Deklaratiivinen metodien kombinointi, ja
- Metaobjektiprotokolla.

CLOSin keskeisiä olioita ovat *luokat*, *ilmentymät*, *geneeriset funktiot* ja *metodit*.

Geneerisillä funktioilla tarkoitetaan funktioita, joiden toiminta riippuu niille annettujen argumenttien tyypeistä. CLOSin geneeriset funktiot "omistavat" *metodeja*, joista funktiokutsussa välitettyjen argumenttien tyyppien perusteella valitaan osa suoritettavaksi. Valinta voi tapahtua useamman kuin yhden argumentin tyypin perusteella. CLOS siis poikkeaa tavanomaisesta oliokeskeisestä ajattelusta jossa metodit kuuluvat olioluokille, ja metodin valinta tapahtuu vain funktion ensimmäisen argumentin tyypin perusteella.

CLOS tarjoaa mahdollisuuden määritellä olioluokkia, käyttää moniperintää tässä määrittelyssä. Olioilla voi olla nimettyjä attribuutteja, *lovia*, joilla voi olla arvo. Luokkamäärittelyssä voidaan spesifioida mikä loven oletusarvo on, millä funktioilla loven arvoa voidaan muuttaa tai se voidaan lukea, ja miten loven alkuarvo spesifioidaan olioilmentymää luotaessa.

Geneerisen funktion kutsussa voidaan suorittaa useita metodeja. Tapa, jolla metodit ja niiden suoritusjärjestys valitaan, on nimeltään *metodikombinointi*. Tavalliset, ns. *primary*-metodit voivat joko peittää ja spesialisoida yläluokassa annetun metodin. Lisätoimintoja voidaan toteuttaa ennen ja jälkeen varsinaisen

metodin ajoa suoritettavilla *before*- ja *after*-metodeilla. Mikäli halutaan varmistua että määriteltävä metodi ajaa aina ennen myöhemmin määriteltyjä before-metodeja tai myöhemmin määriteltyjen after-metodien jälkeen, voidaan käyttää ns. *around*-metodityyppiä.

Metaobjektiprotokollalla tarkoitetaan dokumentoitua geneeristen funktioiden ja luokkien kokoelmaa, joka "aukaisee" oliojärjestelmän toteutuksen laajennuksia varten. Kyse on siitä, että oliojärjestelmä on itse toteutettu oliokeskeisenä ohjelmana, ja vieläpä itsellään.

1.2. Metaobjektiprotokollat

Metaobjektiprotokollien avulla pyritään ohjelmointikielten suunnittelun iäisyyskysymys "eleganssi vastaan tehokkuus" ratkaisemaan siten, että molemmat tavoitteet toteutuvat. Metaobjektiprotokollat tarjoavat ohjelmoijille mahdollisuuden inkrementaalisesti laajentaa ohjelmointikieltä. Periaatteessa mihin tahansa olio-ohjelmointikieleen voitaisiin toteuttaa metaobjektiprotokolla.

Tässä kirjoituksessa metaobjektiprotokollalla tarkoitetaan mm. sitä, että oliokeskeinen ohjelmointikieli tai muu oliojärjestelmä on toteutettu oliokeskeisenä ohjelmana, ja vieläpä itsellään[1]. Mikäli kielen oliokeskeinen toteutus on dokumentoitu ja laajennettavissa, puhutaan metaobjektiprotokollasta.

Metaobjektiprotokollilla on seuraavia hyviä ominaisuuksia:

- **Kannettavuus** paranee, koska ohjelmoijille on tarjolla standardoitu ja dokumentoitu tapa laajentaa ohjelmointikieltä, eikä heidän tarvitse turvautua vaikeasti ylläpidettäviin "temppuihin".
- **Laajennusten hallinta** paranee, koska metaobjektiprotokollien avulla laajennusten vaikutusalue saadaan rajattua tarkasti (ts. standardikäyttäytyminen ei muutu).
- **Laajennusten yhdistäminen** tulee mahdolliseksi, ts. yksi laajennus ei tee toisen käyttöä mahdottomaksi.
- **Tehokkuuden lisääminen** tulee mahdolliseksi mm. erilaisten *caché*-tekniikoiden hallitun käyttöönoton avulla.

1.3. BEEF

BEEF (BOSS's Extremely Elegant Frame System, [Lassila 91]) on Common Lispillä kirjoitettu kehysjärjestelmä, joka kehitettiin Teknillisen korkeakoulun Tietojenkäsittelyopin laboratoriossa alunperin korvaamaan CRL-kehyskieli (Carnegie Representation Language, [Carnegie 86]), ja täten mahdollistamaan erään ohjelmiston siirtäminen TI Explorer -lisp-työasemasta tavalliseen Macintosh II -työasemaan. Aluksi

---

[1] Erityisesti Lisp-maailmassa kielen introspektiivisyys ja "metasirkulaariset" määritelmät eivät ole vieraita.

BEEFiä käytettiin mallitustehtäviin, mutta sittemmin sitä on kehitetty oliokeskeisen ohjelmoinnin suuntaan ja käytetty perusohjelmointityökaluna täydentämässä Common Lispiä. BEEF on tehokas ja sekä staattiselta että dynaamiselta muistinkulutukseltaan melko pieni. Koska se on kirjoitettu kokonaan Common Lispillä, on se myös kannettava.

BEEF on käsitteellisesti selkeä ja perustuu pieneen joukkoon peruskäsitteitä. *Kehykset* ovat BEEFin varsinaisia olioita. Ohjelmoijan kannalta kehystä edustaa kehyksen nimeävä symboli. Tämä helpottaa asioita erityisesti mallitustehtävissä. Lisp-järjestelmien muistinhallinta huomioonottaen voidaan käyttää myös nimettömiä kehyksiä, jotka roskienkeruun on helpompi vapauttaa.

Kehyksillä on *lovia* joita kehykset voivat omistaa ja periä muilta kehyksiltä. Omien ja perittyjen lovien välillä ei ole käsitteellisesti eroa. Normaalisti perintä tapahtuu luokkarakennetta noudatellen, mutta sitä voidaan myös muuttaa. Lovilla voi olla mielivaltainen määrä *arvoja*, jotka kulkeutuvat perinnässä kehykseltä toiselle. Loven arvojen muuttaminen heijastuu sen periviin kehyksiin.

Kehysten välille (ja myös kehysten ja muun lisp-datan välille) voidaan muodostaa *relaatioita*, joita voidaan käyttää hakualgoritmien toteutuksessa ja joilla voidaan myös muuttaa perintäpolkuja. Lovet ajatellaan relaatioiksi kehysten ja arvojensa välille. Muodostettaessa monimutkaisempia relaatioita kuvataan kehysten välinen loviketju erityisen polkukieliopin lausekkeiden avulla.

Loville voidaan määritellä *käänteislovi*, jolloin loven päivittäminen aiheuttaa käänteisloven päivittämisen loven arvona olevissa kehyksissä. Tämä mahdollistaa kehysverkon konsistenssin vaivattoman ylläpidon. Lovien ja relaatioiden kuvaukset ovat kehyksiä; niitä voidaan käsitellä BEEFin omilla primitiiveillä.

Mahdollisuus periä usealta luokalta on hyödyllinen ominaisuus missä tahansa oliokeskeisessä järjestelmässä. Mallitustehtävissä siitä on suurta hyötyä, koska se mahdollistaa olion esittämisen useasta näkökulmasta (esim. fyysisestä ja taloudellisesta). Moniperintää käyttäen voidaan myös määritellä lisäominaisuuksia, joita sitten voidaan "sekoittaa" varsinaisiin olioluokkiin. BEEF tukee moniperintää, ja lisäksi voidaan luokkahierarkiaan (**is-a** -relaatioihin) rajoittuva perintä sivuuttaa ja käyttää mielivaltaista relaatiolauseketta määrittelemään joidenkin ominaisuuksien perintä.

Ohjelmoijan määrittelemien relaatioiden käyttö on kätevää haluttaessa saada tieto siirtymään kehyksestä toisiin, erityisesti kun muistetaan, että kehysjärjestelmissä perintää tapahtuu ajoaikana. Perinteisissä oliokeskeisissä järjestelmissähän perintä on lähinnä määrittelyaikainen tapahtuma.

Maailmat mahdollistavat kehysjärjestelmän useiden samanaikaisten tilojen ylläpidon. Monimutkaisia hakuja toteutettaessa havaitaan monasti, että hakuavaruuden tilat ovat erittäin kompleksisia; kenties koko järjestelmän tila on hakupuun solmu. Tällöin haun edetessä on hankalaa pitää kirjaa tehdyistä ja perutuista muutoksista. Puumaisesti järjestyvät maailmat tarjoavat automaattisen mekanismin muutosten perumiseen. Maailman tuhoaminen peruu kaikki kyseiseen maailmaan sen luomisen jälkeen tehdyt muutokset. Tehtyjä muutoksia voidaan myös vyöryttää ylöspäin maailmahierarkiassa.

Jotta BEEFillä voitaisiin todella ohjelmoida eikä vain mallittaa, on siihen rakennettu mahdollisuus helposti määritellä metodeja eli *viestien käsittelijöitä*. Metodit toimivat syntaktisesti samaan tapaan kuin CLOSissa: jokaista metodin nimeä (l. *viestiä*) vastaa yksi diskriminointifunktio, jota kutsutaan ja joka päättää minkä kehyksen metodi viestin todella käsittelee. Metodit on talletettu kehyksiin ja diskriminaatio tapahtuu vain funktion ensimmäisen argumentin tyypin perusteella. Viestien diskriminointifunktiot voidaan myös määritellä *yleistetyiksi muuttujaviittauksiksi* (engl. generalized variable references), ts. niille voidaan määritellä erityinen **setf**-metodi. Viestin käsittelijän sisällä vastaanottajakehyksen lovet näkyvät yksinkertaisina muuttujaviittauksina.

2. CLOSin metaobjektiprotokolla

Common Lisp Object Systemin määrittelyn viimeinen osa, "The Common Lisp Object System Meta-Object Protocol", ei tule olemaan osa tulevaa ANSI Common Lisp -standardia. Metaobjektiprotokollan määritellyt ja kehittänyt tutkimusryhmä on kuitenkin julkaissut kirjan, josta on muodostunut jonkinlainen de facto standardi CLOSin metaobjektiprotokollaa varten [Kiczales et al 91]. Useat Common Lisp -valmistajat ovat jo menneet osittain tai kokonaan tämän standardin taakse. Tässä kirjoituksessa tästä metaobjektiprotokollasta käytetään nimitystä CLOS MOP.

2.1. Aliprotokollat

CLOS MOP on jaettu useisiin kerroksiin tai aliprotokolliin, joista tärkeimmät ovat

- **Metaobject Initialization Protocol**, *metaolioiden alustusprotokolla*, joka määrittelee miten metaoliot luodaan ja saatetaan tietoisiksi toisistaan.
- **Class Finalization Protocol**, *luokkien viimeistelyprotokolla*, joka määrittelee miten luokkaoliot saatetaan siihen tilaan että niistä voidaan luoda ilmentymiä.
- **Instance Structure Protocol**, *ilmentymien rakenneprotokolla*, joka määrittelee miten ilmentymien lovien arvoja käsitellään.
- **Generic Function Invocation Protocol**, *geneeristen funktioiden kutsuprotokolla*, joka määrittelee miten geneerisiä funktioita kutsutaan.
- **Dependent Maintenance Protocol**, *riippuvuuksien ylläpitoprotokolla*, joka määrittelee miten eri metaoliot varmistuvat säilyttämänsä tiedon eheydestä. Tehokkuussyistä nimittäin samaa informaatiota saatetaan eri muodossa säilyttää monessakin eri paikassa, ja tiedon alkuperäisen lähteen muuttuessa on muutokset voitava päivittää kaikkialle.

Tässä työssä keskitytään näistä protokollista kolmeen ensimmäiseen; geneeristen funktioiden ja metodien määrittely ja kutsuminen jätetään vähemmälle huomiolle.

CLOS MOP perustuu funktionaalisiin ja proseduraalisiin aliprotokolliin. Funktionaalisella protokollalla tarkoitetaan funktiota, jota käytetään laskemaan jokin tulos; tämä tulos talletetaan ja jokin toinen osa

protokollaa käyttää sitä jossakin muualla[2]. Proseduraalisella protokollalla taas tarkoitetaan sitä, että jokin funktio suoraan aikaansaa jonkin tietyn käyttäytymisen sillä hetkellä kun sitä käytetään.

2.1.1. Metaolioiden alustusprotokolla

Tämä protokolla määrittelee miten metaoliot luodaan ja saatetaan tietoisiksi toisistaan. Metaolioita ovat luokat, lovimäärittelyt, geneeriset funktiot, metodit ja metodikombinaatiot. Kaikki metaoliot ovat ilmentymiä ns. metaluokista. CLOS MOP määrittelee standardiversiot kaikista eri metaluokista. Näitä ovat

`standard-class`	Kaikkien `defclass`-makrolla luotavien luokkien oletus-metaluokka.
`standard-object`	Ilmentymä luokasta `standard-class`, kaikkien ilmentymien (paitsi itsensä) yläluokka, tyyppijärjestelmän perusluokan t välitön aliluokka.
`standard-slot-definition`	Kaikkien lovimäärittelyjen yläluokka. Oletusluokkia näille määrittelyille ovat `standard-slot-definition` -luokan välittömät aliluokat `direct-slot-definition` ja `effective-slot-definition`.
`standard-generic-function`	Kaikkien geneeristen funktioiden oletusluokka.
`standard-method`	Kaikkien metodien oletusluokka.
`method-combination`	Kaikkien metodikombinaatioiden yläluokka.

Näistä kaikki paitsi `standard-slot-definition` ovat CLOSin itsensä määrittelemiä (siis standardoituja) luokkia ja kuuluvat CLOS MOPin ulkoiseen rajapintaan. Koska metaoliot ovat ilmentymiä aivan kuin "tavallisetkin" oliot, voidaan metaolioita luoda geneerisellä funktiolla `make-instance`.

Luokkaolioiden luonti tapahtuu funktion `ensure-class` avulla. Määrittelymakro `defclass` laajenee kutsuksi tähän funktioon, mutta sitä voi myös kutsua suoraan. Modifioitava rajapinta on `ensure-class` -funktion kutsuma geneerinen funktio `ensure-class-using-class`, joka puolestaan kutsuu joko geneeristä funktiota `make-instance` (mikäli luokka luodaan ensimmäistä kertaa) tai geneeristä funktiota `reinitialize-instance` (mikäli luokka oli jo olemassa).

Seuraavat alustusparametrit määrittelevät luokan ominaisuudet:

`:direct-default-initargs`	Lista alustusparametrejä, joilla peitetään perittyjen luokkien oletusalustusarvoja.

---

[2] Tulosten etukäteen tapahtuvasta laskemisesta ja tallettamisesta, siis eräänlaisesta *caché*-tekniikasta, käytetään englanninkielistä termiä *memoization*.

`:direct-slots`	Luokan määrittelemät uudet lovet (ns. “kanonisoituja lovimäärittelyjä”).
`:direct-superclasses`	Lista luokan välittömien yläluokkien nimistä.
`:name`	Luokan nimi.

Nämä alustusparametrit vastaavat melko suoraan `defclass`-makrossa annettavia tietoja, ja suoraan `ensure-class` -funktion parametrejä.

Lovimäärittelyolioiden luonti tapahtuu aina `make-instance`lla. Loven uudelleenmäärittely ei ole mahdollista (`reinitialize-instance` signaloi virheen). Lovimäärittelyoliot luodaan kanonisoitujen lovimäärittelyjen perusteella.

Seuraavat alustusparametrit määrittelevät loven ominaisuudet:

`:name`	Loven nimi.
`:initform`	Loven oletusarvolauseke.
`:initfunction`	Parametriton funktio, jonka suoritus evaluoi oletusarvolausekkeen oikeassa leksikaalisessa sidontaympäristössä.
`:type`	Loven arvojen tyyppi.
`:allocation`	Loven tallennustapa: `:instance` ilmentymäloville ja `:class` luokkaloville.
`:initargs`	Lista loven alustusparametrien nimistä.

Geneeristen funktioiden luonti ja metodien luonti tapahtuu vastaavasti. Annetut alustusparametrit vastaavat sitä informaatiota jota on mahdollista antaa `defgeneric` ja `defmethod` -määrittelymakroissa.

2.1.2. Luokkien viimeistelyprotokolla

Tämä protokolla määrittelee, miten luokkaoliot saatetaan siihen tilaan, että niistä voidaan luoda ilmentymiä. Luokan viimeistely l. “finalisointi” kerää luokkaolioon kaiken sen informaation jonka luokka voi periä yläluokiltaan. Viimeistely suoritetaan jossakin vaiheessa luokan määrittelyn ja ensimmäisen ilmentymän luonnin välillä. Tämä protokolla on tarpeen siksi, että luokkamäärittelyissä voi olla viittauksia toisiin luokkiin, joita ei vielä ole määritelty, ja siksi perintää ei voida suorittaa luokkaolion alustuksen yhteydessä.

Viimeistely suoritetaan geneerisellä funktiolla `finalize-inheritance`. Luokan presedenssilista (class precedence list, perittyjen yläluokkien järjestetty lista) lasketaan geneerisellä funktiolla `compute-class-precedence-list` ja talletetaan luokkaolioon. Luokan lovet lasketaan geneerisellä funktiolla `compute-slots`, ja talletetaan luokkaolioon. Funktio `compute-slots` puolestaan kutsuu geneerisiä funktioita `effective-slot-definition-class` ja `compute-effective-slot-definition`.

2.1.3. Ilmentymien rakenneprotokolla

Tämä protokolla määrittelee miten ilmentymien lovien arvoja käsitellään. Protokolla määrittelee toimintamekanismin CLOSin lovienkäsittelyfunktioita varten. Jokaista lovenkäsittelyfunktiota varten on olemassa geneerinen funktio, joka saa argumenteikseen ilmentymän ja loven lisäksi myös ilmentymän luokan; tämä siksi, että metodit voitaisiin spesialisoida eri metaluokkia varten.

Seuraavassa on lueteltu lovenkäsittelyfunktiot ja niitä vastaavat geneeriset funktiot::

`slot-boundp`	`slot-boundp-using-class`
`slot-makunbound`	`slot-makunbound-using-class`
`slot-value`	`slot-value-using-class`
`(setf slot-value)`	`(setf slot-value-using-class)`

Kannettavat CLOS MOP -laajennukset voivat vaikuttaa ilmentymälovien käsittelyyn (niihin, joiden `:allocation` -alustus on `:instance`).

2.2. Toteutuksia

Useat Common Lisp -valmistajat ovat jo menneet CLOS MOPin taakse ja tukevat sitä joko kokonaan tai osittain. Kirjoittajan kokeilemista Common Lisp -järjestelmistä CLOS MOPia tuki Franz Allegro Common Lisp liki kokonaan, ja Macintosh Common Lisp osittain.

CLOS MOPin kehittäjien kirjassa [Kiczales et al 91] esitetään Closette-niminen suppea CLOS-toteutus, joka tukee (kirjan tutoriaaliosan vaatimassa määrin) CLOS MOPia.

Myös Xeroxin public-domain -toteutus CLOSista, Portable Common Loops, tukee metaobjektiprotokollaa.

3. Macintosh Common Lisp ja sen MOP

Macintosh Common Lisp (MCL) on Apple Computer, Inc:n toteuttama Common Lisp -ohjelmointiympäristö [Apple 91]. MCL versio 2.0 on täysi Common Lisp ja tukee nimenomaan "second edition" -standardiehdotusta. MCL toteuttaa osan (n. 30 %) CLOS MOP -määrittelyn funktiorajapinnasta. Applen ilmoituksen mukaan vasta seuraavassa versiossa (versio 2.1, ilmestyy ehkä vuonna 1993) harkitaan täyttä metaobjektiprotokollaa [StClair 92].

3.1. Olemassaolevat piirteet

Osa MCL:n toteuttaman metaobjektiprotokollan funktioista kuuluu myös CLOSin standardoituun määrittelyyn. Toteutetut funktiot ovat lähinnä luonteeltaan "introspektiivisiä", ts. eri asioiden lukemiseen liittyviä. Varsinaisia muutoksia ja laajennuksia on ao. protokollan avulla vaikea tehdä. Useiden funktioiden toiminta ei kuitenkaan sisäisesti noudata metaobjektiprotokollaa, ainoastaan niiden ulkoinen kutsurajapinta. Tässä kirjoituksessa MCL:n metaobjektiprotokollasta käytetään nimitystä MCL MOP.

Funktiot jakautuvat karkeasti kolmeen eri luokkaan:

- **Luokka- ja lovimäärittelyolioiden tarkastelu**. Luokat ovat MCL MOPissa ilmentymiä luokasta `standard-class`, mutta lovimäärittelyille ei ole omaa luokkaa, vaan ne on esitetty listoina. Seuraavat funktiot on toteutettu:

  `class-direct-subclasses`

  `class-direct-superclasses`

  `class-name`

  `class-precedence-list`

  `class-prototype`

  `slot-definition-name`

- **Metodit ja geneeriset funktiot**. Seuraavat funktiot on toteutettu:

  `ensure-generic-function`

  `add-method`

  `remove-method`

  `generic-function-method-class`

  `generic-function-method-combination`

`generic-function-methods`

`compute-applicable-methods`

`compute-effective-method`

`find-method-combination`

`method-function`

`method-generic-function`

`method-qualifiers`

`method-specializers`

`specializer-direct-generic-functions`

`specializer-direct-methods`

- **Ilmentymien luonti** toimii MCL:ssä luonnollisesti täysin CLOSissa määritellyllä tavalla (ts. geneerisiä funktioita `initialize-instance`, `reinitialize-instance`, `shared-initialize`, jne. kutsutaan normaalisti). CLOS MOPin erikseen mainitsemista funktioista seuraavat ovat osa MCL MOPia:

  `allocate-instance`

  `make-instance`

3.2. Metaobjektiprokollan puuttuvien piirteiden lisäys MCL:iin

MCL:n olemassaolevien funktioiden toiminnan muuttamiseksi on kirjoitettu määrittelymuoto `define-overridden-function`, jolla funktion vanha määritelmä voidaan ottaa talteen, määritellä uusi funktio vanhalle nimelle, ja myöhemmin alkuperäistä funktiota voidaan kutsua.

3.2.1. Metaolioiden alustusprotokolla

Metaolioiden alustusprotokollan osalta MCL MOPiin lisättiin seuraavat funktiot:

`slot-definition-allocation`	Toteutettu spesialisoimattomana metodina. Palauttaa aina arvon `:instance`.
`slot-definition-initargs`	Toteutettu spesialisoimattomana metodina. Hakee lovi-määrittelyä edustavasta listarakenteesta oikean elementin.
`slot-definition-initform`	Toteutettu kuten `slot-definition-initargs`.
`slot-definition-initfunction`	Toteutettu kuten `slot-definition-initargs`.
`slot-definition-name`	Toteutettu kuten `slot-definition-initargs`.
`slot-definition-type`	Toteutettu kuten `slot-definition-initargs`.

`ensure-class`	Toteutettu peittämällä MCL:n sisäinen funktio `ccl::%defclass`.
`ensure-class-using-class`	Funktio `ensure-class` kutsuu tätä geneeristä funktiota. MCL MOP -toteutuksessa tähän funktioon piti lisätä muutama uusi avaisanaparametri, sillä alkuperäinen `ensure-class-using-class` -funktion argumentiilista ei vastaa täysin `ccl::%defclass` -funktion argumenttilistaa. Uusia parametrejä ovat `:class-slots` (sillä MCL erottaa toisistaan luokka- ja ilmentymälovet) sekä `:metaclass`.

Oikeiden lovimäärittelymetaolioiden lisääminen protokollaan osoittautui liian vaikeaksi, sillä lovien luonnin osalta MCL:llä ei ole edes sisäistä protokollaa. Näin ollen tyydyttiin ainoastaan lovimäärittelyjen tarkastelemiseen.

3.2.2. Luokkien viimeistelyprotokolla

Luokkien viimeistelyprotokollan osalta toteutettiin seuraavat geneeriset funktiot:

`finalize-inheritance`	Yritetty toteuttaa peittämällä MCL:n sisäinen funktio `ccl::initialize-class-and-wrapper`; tämä funktio kutsuu sisäistä rekursiivista funktiota eikä rekursiivinen kutsu mene funktiosymbolin kautta, joten tämän funktion kutsun jälkeen suoritetaan rekursiivinen `finalize-inheritance` -metodin kutsusarja.
`compute-class-precedence-list`	Toteutettu yksinkertaisesti peittämällä MCL:n sisäinen funktio `ccl::compute-cpl`; vanha funktio voidaan peittää täydellisesti.
`class-finalized-p`	Tieto onnistuneesta viimeistelystä saadaan tutkimalla luokkaolion `ccl::own-wrapper` -loven sidontaa.
`validate-superclass`	Kirjoitettu kokonaan CLOS MOP-standardin kuvauksen mukaisesti.

Geneerinen funktio `finalize-inheritance` on erittäin vaikea toteuttaa alkuperäisen funktion `ccl::initialize-class-and-wrapper` optimoidun toteutuksen takia. Toteutus on seuraava:

```
(defmethod finalize-inheritance ((class class)
                                 &optional (finalizedp
                                            (class-finalized-p class)))
  (unless finalizedp
    (dolist (s (class-direct-superclasses class))
      (unless (validate-superclass class s)
        (error "Class ~S is an incompatible superclass of ~S" s class))
      (finalize-inheritance s))))
```

Kun `finalize-inheritance` tulee kutsutuksi, on MCL jo alustanut luokkaoliot (rekursiivisesti). Superluokkapuussa suoritetaan siis kaksi läpikäyntiä. Effektiivisesti luokkien viimeistely kuitenkin toimii oikein. Peitetty funktio `ccl::initialize-class-and-wrapper` on kirjoitettu seuraavasti:

```
(define-overridden-function ccl::initialize-class-and-wrapper nil (class)
  (let ((finalizedp (class-finalized-p class)))
    (prog1 (ccl::initialize-class-and-wrapper class)
      (finalize-inheritance class finalizedp))))
```

3.2.3. Ilmentymien rakenneprotokolla

MCL:n `slot-value` -funktion toiminta perustuu sisäiseen funktioon `ccl::%slot-value`, joka palauttaa suoraan sen (raa'an) arvon joka ilmentymän loveen on talletettu. Tämä arvo voi olla joko todellinen Common Lisp -olio, tai jokin seuraavista vakioista (jotka siis eivät ole Common Lispin tyyppijärjestelmän alaisia arvoja):

`(ccl::%unbound-marker-8)`	Makro, jonka laajennustulos on "unbound" -merkki (edustaa lovea, jolla ei ole arvoa, eli joka on "unbound").
`(ccl::%slot-missing-marker)`	Makro, jonka laajennustulos on "slot missing" -merkki (edustaa puuttuvaa lovea).

Näistä makroista, ja funktiosta `ccl::%slot-value`, on MCL MOP -toteutuksessa (ylläpitosyistä) tehty makrot `primitive$unboundp`, `primitive$missingp` ja `primitive$slot-access`; näiden avulla geneerinen funktio slot-value-using-class voidaan kirjoittaa seuraavasti:

```
(defmethod slot-value-using-class ((class ccl::class) object slot)
  (let ((value (primitive$slot-access object slot)))
    (cond ((primitive$unboundp value)
           (slot-unbound class object slot))
          ((primitive$missingp value)
           (slot-missing class object slot 'slot-value))
          (t value))))
```

Virheet siis annetaan CLOSin virhemetodeilla `slot-unbound` ja `slot-missing`. Nämähän voidaan määritellä tarvitaessa uudestaan spesialisoiduille luokille.

Alkuperäinen `slot-value` -funktio on määritelty uudelleen seuraavasti:

```
(defun slot-value (object slot-name)
  (slot-value-using-class (class-of object) object slot-name))
```

Alkuperäistä funktiota ei tarvita, koska metodi `slot-value-using-class` toteuttaa sen toiminnot.

4. BEEFin muuttaminen CLOS-ohjelmaksi

BEEFin keskeiset CLOSin kanssa epäyhteensopivat mekanismit ovat luokkien määrittely, lovien käsittely ja metodit. Nämä tulee CLOS-yhteensopivassa BEEF-toteutuksessa modifioida käyttäen metaobjektiprotokollaa. Koska BEEFin funktionaliteetti ylittää CLOSin funktionaliteetin, on metaobjektiprotokollaa myös jonkin verran laajennettava.

4.1. Luokkien määrittely

Kehysjärjestelmän ero CLOSin kaltaiseen oliojärjestelmään on mm. se, että ero normaalitason ja metatason välillä on kehysjärjestelmissä monasti hämärtynyt. Näin on asia myös BEEFissä. Luokkaoliot eivät siis ole vain metaolioita, vaan myös normaaleja (lue: "sovellusohjelmien käsittelemiä") olioita. Jos siis CLOSissa määritellään luokka `A` jonka metaluokka on `META-A`, on itse luokkaolio vain tyyppiä `META-A`, mutta BEEFissä sen tulisi olla sekä tyyppiä `A` että `META-A`.

Ongelma voidaan ratkaista esimerkiksi käyttämällä kahta eri oliota edustamaan luokkaa: itse luokkaolio (CLOSin metaolio) syntyy luokkamäärittelyn tuloksena. Se puolestaan luo jonkinlaisen luokkaprototyypin kaltaisen olion, joka on ilmentymä ao. luokasta. Tätä oliota käytetään edustamaan luokkaa kehysjärjestelmässä.

4.2. Lovien käsittely

Lovien toiminta on se piirre, jossa BEEFin funktionaliteetti selvimmin ylittää CLOSin funktionaliteetin. BEEF-olioiden lovet ovat herkkiä BEEFin hierarkkiselle maailmajärjestelmälle, niiden perintä on dynaamista, niillä voi olla automaattisia käänteislovia, ja niihin voidaan asentaa demoneja. Kaikki nämä piirteet voidaan kuitenkin toteuttaa CLOS MOPin avulla.

4.2.1. Maailmaherkät lovet

BEEFin nykyisessä toteutuksessa maailma/arvo -parit ovat assosiatiivisessa tietorakenteessa, joka on talletettu kehyksen loveen. Arvoa luettaessa tietorakenteesta assosioidaan maailman perusteella, ja tarvittaessa toistetaan haku ylöspäin maailmapuussa (kohti juurta).

Maailmaherkät lovet toteutetaan siten, että CLOS-ilmentymän tallettama arvo on maailma/arvo -parien tietorakenne, ja maailmaherkkien lovien lovimäärittelyluokka spesialisoi `slot-value-using-class` -metodin, seuraavasti:

```
(defmethod slot-value-using-class ((class frame-class)
                                   (object frame)
                                   (slot world-slot))
  (find-value-in-world *world* (call-next-method)))
```

Tässä oletetaan, että vallitseva maailma on talletettu muuttujaan `*world*` (kuten BEEFin nykyisessä versiossa), että `find-value-in-world` on maailman haku maailma/arvo -tietorakenteesta, ja että määritellyn metodin peittämä metodi palauttaa mitä ikinä loveen onkaan talletettu.

4.2.2. Dynaaminen perintä

Dynaamiset lovet toteutetaan myös peittämällä `slot-value-using-class` -metodi, käyttäen BEEFin nykyistä dynaamista perintähakua. Haku siis etenee kehyksestä toiseen halutussa perimysjärjestyksessä; jokaisen kehyksen paikallisiin arvoihin päästään käsiksi kutsumalla peitettyä metodia (funktiolla `call-next-method`).

4.2.3. Automaattiset käänteislovet

Automaattiset käänteislovet voidaan toteuttaa `(setf slot-value-using-class)` metodin `:before` ja `:after`-metodeina. Näistä `:before`-metodi poistaa poistettavan loven arvoa vastaavan käänteisviittauksen, ja `:after`-metodi lisää uutta loven arvoa vastaavan viittauksen. Koska käänteisviittaukset voivat olla vain kehyksestä toiseen, vain sellaiset lisättävät arvot jotka ovat kehyksiä kiinnostavat tätä mekanismia.

Myös metodille `slot-makunbound` on määriteltävä spesialisaatio, sillä loven arvojen poistaminenhan luonnollisesti vaikuttaa myös käänteisloveen.

Mikäli on kyse yksiarvoisista lovista, voidaan mekanismi toteuttaa normaaleja `slot-value-...` -operaatioita käyttäen. Moniarvoisia lovia varten tarvitaan taas muutenkin mekanismi arvojen lisäämistä ja poistamista varten.

4.2.4. Demonit

Demonilla tarkoitetaan funktiota joka ajaa loven arvon muuttuessa tai loven arvoa luettaessa. Oliokeskeisessä järjestelmässä on tietenkin olemassa protokolla näitä operaatioita varten, ja spesialisoimalla tietyt metodit voidaan saada aikaan "demonista" toimintaa (varsinaisten lovioperaatioiden `:primary`, `:before` ja `:after` -metodeilla). Joskus kuitenkin halutaan demoneja liittää loviin "lennossa", ts. dynaamisesti.

Tyypillisesti demoneja on kahden- tai kolmenlaisia (loven arvon lukeminen, loven arvon kirjoittaminen, ja mahdollisesti loven arvon poistaminen). Demonilovet toteutetaan siis kolmena eri luokkana, joista jokainen voidaan sekoittaa lovimääritelmään. Esimerkkinä annetaan arvojen lukemisen suhteen "demoninen" lovi:

```
(defclass read-daemon-slot ()
  ((primary-read-daemon
    :initform nil
    :accessor slot-primary-read-daemon)
   (before-read-daemon
    :initform nil
    :accessor slot-before-read-daemon)
   (after-read-daemon
    :initform nil
    :accessor slot-after-read-daemon)))

(defmethod slot-value-using-class ((class frame-class)
                                   (object frame)
                                   (slot read-daemon-slot))
  (let ((daemon (slot-primary-read-daemon slot)))
    (if daemon
      (funcall daemon object slot #'(lambda () (call-next-method)))
      (call-next-method))))

(defmethod slot-value-using-class :before ((class frame-class)
                                           (object frame)
                                           (slot read-daemon-slot))
  (let ((daemon (slot-before-read-daemon slot)))
    (when daemon
      (funcall daemon object slot))))

(defmethod slot-value-using-class :after ((class frame-class)
                                          (object frame)
                                          (slot read-daemon-slot))
  (let ((daemon (slot-after-read-daemon slot)))
    (when daemon
      (funcall daemon object slot))))
```

Lovella on siis erikseen sekä before-, primary- että after-demonit. Before- ja after-demonit ovat muotoa

```
#'(lambda (object slot)
    …)
```

ja ne voivat ainoastaan tehdä joitakin ylimääräisiä toimintoja loven lukemisen aikana, mutta eivät voi vaikuttaa itse luettavaan arvoon. Primary-demoni taas on muotoa

```
#'(lambda (object slot continuation)
    …
    (funcall continuation))
```

jossa kolmas argumentti on funktionaali, jota kutsumalla päästään käsiksi peitettyyn `slot-value-using-class` -metodiin. Primary-demonin palauttama arvo on lopullinen loven luettu arvo. Demonifunktion ei titetenkään ole pakko kutsua funktionaalista parametriaan, mikäli kyseessä on lovi jonka arvot ovat puhtaasti laskennan tuloksia.

4.3. Metodit

BEEFin oma metodimekanismi voidaan kokonaan korvata CLOSin geneerisillä funktioilla ja metodeilla. Tällä tavalla voidaan BEEFiin tuoda entistä voimakkaampi metodimekanismi, ja lisäksi pienentää itse BEEFin toteutusta. Yhteensopivuussyistä BEEFin `defhandler`-määrittelymakro voidaan kirjoittaa uudestaan siten, että se laajenee `defmethod`-kutsuksi.

5. Yhteenveto

CLOS MOP vaikuttaa melko pitkälle harkitulta standardiehdotukselta. Käytännössä CLOS MOPista on muodostumassa jonkinlainen *de facto* standardi, samaan tapaan kuin ensimmäisestä Common Lispin standardiehdotuksesta aikanaan. Tämä mahdollistanee kannettavien CLOS-laajennusten kirjoittamisen. CLOS MOP mahdollistaa myös vanhojen, muilla oliojärjestelmillä kirjoitettujen ohjelmistojen siirtämisen Common Lisp -ohjelmiksi. Esimerkiksi lähteessä [Kiczales et al 91] on esitetty Flavors ja Loops -tyyppisten perintämekanismien toteuttaminen.

MCL MOP on hyvin puutteellinen, mutta pienillä muutoksilla ja lisäyksillä sen funktionaliteetti saadaan tasolle, jolla voidaan jo suorittaa kohtuullista metaobjektiohjelmintia. Tästä on esimerkkinä ehdotus BEEFin CLOS-versioksi tässä työssä.

BEEFin olennaisista osista, kehyksistä, lovista ja metodeista kehyskäsitteen sovittaminen CLOS-maailmaan on kaikkein vaikeinta. Luokkien ja niiden ilmentymien jonkinlainen tyyppiekvivalenssi on vaikea toteuttaa siististi. Tässä työssä ei tarkasteltu mahdollisuutta toteuttaa tyyppien ekvivalenssin tarkastus metodien valinnan aikana, ts. spesialisoimalla esimerkiksi metodit `compute-effective-method` ja `compute-applicable-methods`.

BEEFin lovet siirtyvät CLOS-maailmaan helposti, ja itse asiassa CLOS tarjoaa mahdollisuuden yhdistellä erilaisia lovien ominaisuuksia tarpeen mukaan. Tässä yhteydessä heräsikin ajatus toteuttaa kehysjärjestelmän olennaiset piirteet ainoastaan CLOSin lovienkäsittelyn laajennuksina.

BEEFin metodit voidaan poistaa kokonaan, sillä CLOSin metodimekanismi on vahvempi ja helpokäyttöisempi.

Kaiken kaikkiaan CLOS MOP soveltuu erinomaisesti kehysjärjestelmän rakentamiseen. Se mahdollistaa puuttuvien piirteiden lisäämiseen CLOSiin. Lisäksi se tarjoaa ohjenuoran kehysjärjestelmän oman metaobjektiprotokollan rakentamiseen.

6. Lähdeviitteet

[Apple 91] Apple Computer, 1991. *Macintosh® Common Lisp 2.0 Reference*. Draft 030-5008-A, Cupertino (CA, U.S.A.), Apple Computer, Inc.

[Bobrow 88] Bobrow Daniel G, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales and David A. Moon, 1988. *Common Lisp Object System Specification*. X3J13 Document 88-002R, SIGPLAN Notices, 23 (special issue).

[Carnegie 86] Carnegie Group, 1986. *Knowledge Craft User's Manual*. Pittsburgh (PA, U.S.A.), Carnegie Group, Inc.

[Kiczales et al 91] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow, 1991. *The Art of the Meta–object Protocol*. Cambridge (MA, U.S.A.), MIT Press.

[Lassila 91] Ora Lassila, 1991. *BEEF Reference Manual - A Programmer's Guide to the BEEF Frame System (second version)*. Report HTKK-TKO-C46, Otaniemi (Finland), Laboratory of Information Processing Science, Helsinki University of Technology.

[StClair 92] Bill St Clair, Applen MCL -projektipäällikkö, henk. koht keskustelu sähköpostin välityksellä, maaliskuu 1992.

[Steele 84] Steele Guy L. Jr., 1984. *Common LISP: The Language* (first edition). Hanover (MA, U.S.A.), Digital Press, Digital Equipment Corporation.

[Steele 90] Steele Guy L. Jr., 1990. *Common LISP: The Language* (second edition). Hanover (MA, U.S.A.), Digital Press, Digital Equipment Corporation.

7. Ohjelmalistaukset

Liitteenä on MCL MOP -laajennuksenlähdetiedostot:

"`beef:mop;mop-package.lisp`"	joka määrittelee MCL MOPin nimiavaruuden (ohjelmamodulin) sekä joitakin apufunktioita.
"`beef:mop;mop-for-mcl.lisp`"	joka sisältää itse MCL MOPin koodin, seuraavassa järjestyksessä: • Apufunktioita ja makroja • Lovimäärittelymetaoliot • Ilmentymien rakenneprotokolla • Luokkametaoliot • Luokkien viimeistelyprotokolla