УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Svetlana Jakšić

Types for Access and Memory Control

DOCTORAL DISSERTATION

Светлана Јакшић

Типски системи за контролу меморије и права приступа

ДОКТОРСКА ДИСЕРТАЦИЈА

Нови Сад, 2016

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани запис
Врста рада, **ВР**:		Докторска дисертација
Аутор, **АУ**:		Светлана Јакшић
Ментор, **МН**:		Проф. др Mariangiola Dezani и проф. др Јованка Пантовић
Наслов рада, **НР**:		Типски системи за контролу меморије и права приступа
Језик публикације, **ЈП**:		Енглески
Језик извода, **ЈИ**:		Српски, енгески
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		
Година, **ГО**:		2016
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		6/186/153/53/0/0/0
Научна област, **НО**:		Примењена математика
Научна дисциплина, **НД**:		Формални модели у рачунарству
Предметна одредница/Кqучне речи, **ПО**:		Типски системи, процесни рачуни, конкурентни системи, дистрибуирани системи, безбедност, приватност, цурење меморије
УДК		
Чува се, **ЧУ**:		Библиотеци Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У тези су разматрана три проблема. Први је администрација и контрола права приступа података у рачунарској мрежи са XML подацима, са нагласком на безбедости посматраних података. Други је администрација и котрола права приступа подацима у рачунарској мрежи са RDF подацима, са нагласком на приватности посматраних података. Трећи је превенција грешака и цурења меморије, као и грешака у комуникацији генерисаним програмима написаних на језику Sing# у којима су присутни изузеци. За сва три проблема биће предложени формални модели и одговарајући типски системи помоћу којих се показује одсуство неповољних понашања тј. грешака у мрежама односно програмима.
Датум прихватања теме, **ДП**:		21.01.2016
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	др Силвиа Гилезан, редовни професор
	Члан:	др Мирослав Поповић, редовни професор
	Члан:	др Luca Padovani, ванредни професор
	Члан:	др Јелена Иветић, доцент
	Члан, ментор:	др Mariangiola Dezani, редовни професор
	Члан, ментор:	др Јованка Пантовић, редновни професор

Потпис ментора

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	PhD thesis
Author, **AU**:	Svetlana Jakšić
Mentor, **MN**:	Prof. dr Mariangiola Dezani and prof. dr Jovanka Pantović
Title, **TI**:	Types for Access and Memory Control
Language of text, **LT**:	English
Language of abstract, **LA**:	Serbian, English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	
Publication year, **PY**:	2016
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6/186/153/53/0/0/0
Scientific field, **SF**:	Applied Mathematics
Scientific discipline, **SD**:	Formal Models in Computer Science
Subject/Key words, **S/KW**:	Type systems, process calculi, concurrent systems, distributed systems, security, privacy, memory leaks
UC	
Holding data, **HD**:	Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, **N**:	
Abstract, **AB**:	Three issues will be elaborated and disussed in the proposed thesis. The first is administration and control of data access rights in networks with XML data, with emphasis on data security. The second is the administration and control of access rights to data in computer networks with RDF data, with emphasis on data privacy. The third is prevention of errors and memory leaks, as well as communication errors, generated by programs written in Sing # language in the presence of exceptions. For all three issues, there will be presented formal models with corresponding type systems and showed the absence of undesired behavior i.e. errors in networks or programs.
Accepted by the Scientific Board on, **ASB**:	January 21, 2016
Defended on, **DE**:	

Defended Board, **DB**:	President:	dr Silvia Gilezan, full professor	
	Member:	dr Miroslav Popović, full professor	
	Member:	dr Luca Padovani, associate professor	Menthor's sign
	Member:	dr Jelena Ivetić, assistant professor	
	Member, Mentor:	dr Mariangiola Dezani, full professor	
	Member, Mentor:	dr Jovanka Pantović, full professor	

Acknowledgements

To be honest, at the beginning of my postgraduate studies I had no idea what is ahead of me. I certainly did not expect to meet all the wonderful people and visit all the interesting places to which this adventure led me.

This thesis is the result of years of work with remarkable people who I had the honour to meet and who were my greatest inspiration. Most of all I am thankful to my mentors, Professor Mariangiola Dezani-Ciancaglini and Professor Jovanka Pantović, for their guidance and help. I truly believe that their virtues and goodness are above all the results we achieved. The work on the thesis is finished, but their unique human qualities are something I hope to achieve. To, at least partly, inspire other people the way they do, is my future work.

The union of sets of my co-authors, co-workers and friends is quite wide and their intersection is not empty. I thank all of them for accepting, encouraging, understanding and teaching me in many ways and for letting me experience all significant and unforgettable moments we had.

Without unselfish and infinite support of my family, nothing of this would ever happen. My most sincere gratitude for everything they did for me belongs to them. And yes, “dotkorat” is finally completed.

Rezime

Današnji računarski sistemi se sastoje od umreženih računara, višestruke aplikacije se izvršavaju na jednom računaru, a veb sajtovi imaju više korisnika. Zatim, više procesora, kao i višejezgarni procesori mogu istovremeno vršiti operacije na jednom računaru, a programi se mogu razdvojiti na osnovne delove i na taj način ubrzano izvršavati. Stoga, moramo zaključiti da je *konkurentnost* prisutna svuda. *Komunikacija* i *interakcija* su postale centralni aspekt svih modernih softverskih sistema, polazeći od procesa u geografski distribuiranim mrežama pa do niti procesa koje se izvršavaju na različitim jezgrima u okviru istog procesora. Nedeterminističko preplitanje konkurentnih operacija otežava otkrivanje i otklanjanje grešaka (uzajamnog blokiranja, race conditions, narušavanje bezbednosti,...). Razvoj efikasnih metoda za kontrolu konkurentnosti sistemima, kao što su operativni sistemi i sistemi za rukovođenje bazama podataka, čija se dostupnost i pouzdanost podrazumevaja, je izazovan zadatak. Raznovrsnost i kompleksnost problema vezanih za konkurentnost je prouzrokovala razvoj širokog sprektra *formalnih modela*. Formalni model mora biti jednostavan, ekspresivan, definisan korišćenjem tehnika logike i matematike i mora pružiti sredstva za analizu posmatranog problema. Neki od dobro poznatih modela konkurentnosti su model aktera [74, 4], transakcione memorije [91, 66, 67], Petrijeve mreže [116, 124] i procesni računi [105, 108, 75].

U tezi će biti razmatrana tri problema. Prvi je rukovođenje i kontrola pristupa podacima u distribuiranoj mreži sa polu-strukturiranim podacima u XML formatu, sa naglaskom na pitanja *bezbednosti* posmatranih podataka. Metode za kontrolu prava pristupa, iako sprečavaju neovlašćeni pristup podacima, samostalno nisu dovoljne da u potpunosti zaštite bezbednost podataka. Naime, pristup određenim podacima je ili dozvoljen ili zabranjen, a nemoguće je kontrolisati kako se ti podaci kasnije koriste, kao ni da li je došlo do nepoželjenog uticaja na njih. Zbog toga je neophodno uvesti i druge metode za kontrolu rukovođenja podacima. Primena takvih metoda često može biti složena kada se radi o konkurentnim i distribuiranim sistemima.

Drugi problem je rukovođenje i kontrola pristupa podacima u distribuiranoj mreži sa podacima u RDF formatu, sa naglaskom na pitanja *privatnosti* posmatranih podataka. Potreba da se podaci objavljuju u formatu koji omogućava kombinovanje polu-strukturiranih i strukturiranih podataka prisutna je sve češće. Takvi podaci su najčešće u RDF formatu koji podržava njihovo povezivanje, kombinovanje i razmenjivanje među različitim aplikacima. Velika vrednost mreže povezanih podataka leži u javnoj dostupnosti. Iako jednostavan i širok pristup povezanim podacima ima značajnih prednosti za njihove korisnike, ne odgovaraju

svi podaci takvom načinu pristupa. Na primer, RDF se često koristi za predstavljanje ličnih podataka sa društvenih mreža. To dovodi do pitanja očuvanja privatnosti povezanih podataka jer nedostatak mehanizama za zaštitu privatnosti često obeshrabruje njihovo objavljivanje. Rešavanje ovog problema zahteva jasno objašnjenje pojma privatnosti koji se u [151] definiše u smislu kontrole pristupa i prenosa podataka. Stoga je, kao i kod prvog problema, pored metoda za kontrolu pristupa, neophodno uvesti i dodatne metode da bi se zaštitila privatnost podataka u mreži.

Treći problem je prevencija *grešaka u memoriji* i *curenja memorije*, kao i *grešaka u komunikaciji* generisanim programima napisanim na jeziku koji podržava razmenu poruka bez kopiranja, a u prisutvu izuzetaka. Razmena poruka je fleksibilna paradigma koja dozvoljava autonomnim entitetima da razmenjuju informacije i da se sinhronizaciju. Izraz „razmena poruka" na neki način sugeriše da poruke prelaze sa jednog entiteta na drugi, iako se najčešće razmenjuju samo njihove kopije. Dok je slanje kopija neizbežno u distribuiranim sistemima, prisustvo jedinstvenog memorijskog prostora omogućava primenu razmene poruka bez kopiranja, pri kojoj se razmenjuju samo pokazivači na poruke. Razmena poruka bez kopiranja i izuzeci su očigledno suprotnosti: sa jedne strane, takva razmena poruka zahteva veoma disciplinovan i kontrolisan pristup memoriji, dok, sa druge strane, izuzeci u opštem slučaju prekidaju normalan tok izvršavanja programa. Prevencija gorepomenutih grešaka u prisustvu izuzetaka zahteva statičku analizu programa, kao i podršku sistema u toku izvršavanja.

Ciljevi teze su formalizacija ovih problema i pronalaženje odgovarajućih metoda za njihovo rešavanje. *Procesni računi* su usredsređeni na komunikaciju, interakciju i sinhronizaciju, te predstavljaju pogodan formalni model za probleme koji su razmatrani u ovoj tezi. *Tipski sistemi* pripadaju statičkim metodama za verifikaciju i nametanje raznih svojstava.

Pregled sadržaja i strukture teze

Teza se sastoji od šest poglavlja. Nakon **uvodnog** poglavlja koje opisuje predmet i ciljeve istraživanja, u **drugom poglavlju** je dato više detalja o formalizmima koji su korišćeni. Naime, drugo poglavlje sadrži osnovne pojmove procesnih računa i tipskih sistema, kao i definiciju π-računa.

Osnova svakog konkurentnog izračunavanja je pojam *procesa*. Koncept procesa i konkurentnost su srž modernih operativnih [132] i distribuiranih [139] sistema. Proces predstavlja instancu programa čije je izvršavanje u toku i koja može istovremeno da obavlja više od jednog zadatka koristeći jedinice konkurentnosti zvane *niti*. Međusobna interakcija procesa, tj. slanje i primanje poruka, vrši se preko medijuma za komunikaciju koji se zovu *kanali*. Milner, Parrow i Walker su početkom devedesetih godina prošlog veka predstavili π-račun [108] kao formalni model za konkurentnost. Proces u π-računu je apstrakcija autonomnog entiteta koji može da stupa u interakciju sa drugim procesima, a kanal u π-računu je apstrakcija komunikacione veze između dva procesa. Pomoću π-računa se mogu opisati konkurentni sistemi koji mogu menjati konfiguraciju tokom izvršavanja,

slanjem i primanjem kanala preko kanala. Njegova sintaksa omogućava opisivanje suštine konkurentnih sistema: procesa, paralelne kompozicije procesa, komunikaciju među procesima, prenošenje kanala, kreiranje novih kanala, nedeterminizam i replikaciju procesa. Pojam izračunavanja u procesnom računu može biti dat preko *relacije redukcije* ili preko *sistema označenih tranzicija*. U tezi će biti korišćen prvi način.

Tipski sistemi su jedna od najrasprostranjenijih tehnika za analizu programskih jezika. Oni se koriste za eliminisanje nepoželjnih ponašanja, tj. grešaka koje se javljaju tokom izvršavanja. Svaki jezik ima svoje specifične greške. Generalno govoreći, tipski sistem klasifikuje elemente jezika, koje nazivamo *termi*, na skupove, koje nazivamo *tipovi*, i pokazuje odsustvo određenih grešaka na osnovu dodeljenih tipova. Može se dogoditi da neki termi budu odbačeni od strane tipskog sistema, iako je njihovo ponašanje tokom izvršavanja ispravno. Sa druge strane, precizniji tipski sistem će zaista tipizirati više termova, ali najverovatnije po cenu složenijeg algoritma za proveru tipiziranosti. Stoga je važno napraviti pravi balans između ova dva pitanja.

U konkurentnim programima i sistemima tipski sistemi se često koriste za eliminaciju grešaka kao što su uzajamna blokiranja i race conditions, kao i za proveru disciplinovanog ponašanja, ekstrakciju informacija o programima koja može biti korisna za predviđanje ponašanja programa tokom izvršavanja itd.

Tipski sistemi u ovoj tezi su blisko povezani sa određenim tipskim sistemima za π-račun. Tipski sitemi uvedeni u trećem i četvrtom poglavlju se oslanjaju na *jednostavne tipove* iz šestog poglavlja [130]. Jednostavni tipovi zahtevaju razmenu podataka unapred zadatog tipa. Tipski sistem uveden u šestom poglavlju teze je sličan *tipovima sesija* iz [144, 62]. Osnovna ideja tipova sesija je da se kanali tipiziraju sekvencom tipova koja predstavlja trag njegove upotrebe. Oni obezbeđuju odsustvo grešaka u komunikaciji i poštovanje protokola sesije. Originalni tipovi sesija iz [78, 79] na početku ne prepoznaju razliku između krajeva jednog kanala, dok u [144, 62] autori prave tu razliku koriseći endpoint tipove. U radu [38] se može naći više informacija o sesijama i tipovima sesija, kao i detaljan pregled literature.

Da bismo opisali procesne račune sa tipovima potrebno je uvesti standardnu terminologiju i notaciju, koja je data i u šestom poglavlju [130]. Izraz oblika $u : T$, gde je u ime kanala, a T tip, *dodeljuje* tip imenu. *Tipski kontekst* je konačan skup dodela tipova imenima. Podrazumeva se da su sva imena u jednom kontekstu različita. Tipske kontekste označavamo sa Γ. *Tipski sud* je izraz oblika $\Gamma \vdash e : T$, gde e označava term posmatranog jezika. Tipski sud $\Gamma \vdash e : T$ iskazuje da term e ima tip T u kontekstu Γ. Term je *dobro tipiziran u* Γ ako postoji T takav da se $\Gamma \vdash e : T$ može izvesti iz aksioma i pravila zaključivanja posmatranog tipskog sistema. Tipski sistem je *pouzdan* ukoliko termi koji su dobro tipizirani ne prouzrokuju greške tokom izvršavanja. Termi koji ne prouzrokuju greške tokom izvršavanja se nazivaju *termi sa dobrim ponašanjem*. Da bi se pokazalo da je tipski sistem pouzdan najčešće je potrebno pokazati dva tvrđenja: dobra tipiziranost je očuvana tokom izvršavanja (eng. *subject reduction*) i dobro tipizirani termi imaju dobro ponašanje (eng. *type safety*). Formalizacija ovih tvrđenja zavisi od posmatranog jezika i tipskog sistema, a njihovi dokazi najčešće zahtevaju pokazivanje

više dodatnih osobina.

Treće, četvrto i peto poglavlje razmatraju po jedan od problema navedenih u uvodu i mogu se posmatrati kao odvojene celine koje se oslanjaju na pojmove i račun date u drugom poglavlju. Sva tri poglavlja sadrže definicije odgovarajućih procesnih računa, dobrih ponašanja i tipskih sistema, kao i dokaze pouzdanosti tih tipskih sistema. U narednim paragrafima dajemo pregled rezultata iznesenih u ovim poglavljima i navodimo odgovarajuće publikacije.

Treće poglavlje se odnosi na problem rukovođenja i kontrole pristupa podacima u distribuiranoj mreži sa polu-strukturiranim podacima u XML formatu, sa naglaskom na pitanja *bezbednosti* posmatranih podataka.

Podaci prisutni na vebu su često u formatima fleksibilne strukture koja zavisi od namene samih podataka. Kod takvih, polu-strukturiranih podataka, podaci i njihova šema nisu razdvojeni, kao što je to slučaj kod strukturiranih podataka, prisutnih u relacionim bazama. Polu-strukturirani podaci su lako dostupni, prilagodljivi, prenosivi i pogodni za razmenu između različitih vrsta baza. *Extensible Markup Language* (XML) [19] i XQuery [16, 126] su trenutno standardne metode za izražavanje i pretraživanje polu-strukturiranih podataka.

Jedan od glavnih koraka u rukovođenju distribuiranim sistemima sa podacima je regulacija bezbednosti, koja je jedno od glavnih sredstava za prevenciju neodobrenog pristupa resursima sistema. Upravljanje pristupom na osnovu uloga (eng. role-based access control - RBAC) [129] je metoda za kontrolu pristupa zasnovana na pojmovima korisnika, uloga i dozvola. Ova metoda kontroliše prava pristupa korisnika resursima sistema u skladu sa aktivnostima koje obavljaju. Prilikom pristupa resursima, korisnik ima ona prava koja su dodeljena njegovim ulogama. Uloge u jednom sistemu su statički definisane na osnovu organizacione strukture, te je stoga regulacija bezbednosti svedena na rukovođenje pravima, tj. dozvolama. To čini RBAC jednostavnom i pogodnom tehnikom za kontrolu prava pristupa.

Bezbednost podataka se definiše u smislu održivosti njihove poverljivosti, integriteta i dostupnosti. Osobina poverljivosti razmatra kome se prenose. Integritet podataka se odnosi na neodobrenu modifikaciju podataka i obično je definisan kao očuvanje nekog bitnog svojsta, kao što je konzistentnost, preciznost ili smislenost podataka. U tom smislu, tvrdimo da je bezbedost podataka zaštićena ukoliko korisnici ne prekoračuju prava dodeljena njihovim ulogama, tj.

- ako se podaci prenose samo onim učesnicima kojima je dozvoljeno da im pristupaju, i
- ako procesi pristupaju podacima poštujući unapred zadate politike, i
- ako procesi ne vrše promene na podacima nakon kojih podaci postaju nekonzistentni spram unapred zadatih politika.

U ovom poglavlju razmatramo mrežu odvojenih lokacija koje sadrže XML podatke i procese. U takvoj mreži se može očekivati da nisu svi podaci dostupni svim procesima i da samo ovlašćeni procesi mogu menjati podatke. Zbog toga, podacima i procesima dodeljuju se uloge. Takođe, pretpostavljamo da svaka lokacija

ima svoju *politiku zaštite bezbednosti* koja određuje koje uloge mogu da joj pristupe. Samo procesi kojima je dozvoljeno da u potpunosti pristupe nekim podacima imaju pravo da ih menjaju. Zatim, kako je mogućnost promene prava pristupa jedna od poželjnih karakteristika ovakve mreže, takve promene se moraju vršiti u skladu sa politikama lokacija. Povreda bezbednosti bi bila bilo kakva akcija koja nije dozvoljena ili nakon koje podaci mogu postati nekonzistentni. Nekontrolisano čitanje i prenošenje podatka takođe mogu prouzrokovati povrede bezbednosti. Kao što smo već rekli, da bi se održala bezbednost u sistemu, pored metoda za kontrolu pristupa, neophodno je uvesti i dodatne metode za analizu podataka i procesa.

Peer-to-peer mreže sa dinamičkim veb podacima su modelirane X$d\pi$ računom u radu [55]. Podaci su predstavljeni osnovnim modelom polu-strukturiranih podataka, neuređenim označenim stablom koje sadrži pokazivače na druge delove mreže i neaktivne procese koji nakon pokretanja mogu pretraživati i menjati podatke. Autori su kao model za XML dokumente izabrali stablo sa korenom i označenim granama. U suštini, takvo stablo predstavlja graf u kome su grane označene tagovima XML elemenata. Prilikom izbora modela podataka, autori su odlučili da se pokazivači i neaktivni procesi nalaze samo u listovima stabla. Oba ova izbora, iako se neznatno razlikuju od standardnog modela za XML podatke, nisu uticali na ideje iznete u [55].

Sa ciljem da se RBAC primeni na model peer-to-peer mreže sa polu--strukturiranim veb podacima, u radu

- Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović: Types for Role-based Access Control of Dynamic Web Data in Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10), LNCS, Vol. 6559, pages 1-29, Springer, 2011. ([40])

uvodimo ®X$d\pi$ kao njen formalni model. Bezbednost takve mreže kontrolišemo pomoću tehnika RBAC i tipskog sistem. U našem modelu, mreža je paralelna kompozicija lokacija, pri čemu svaka lokacija sadrži stablo podataka čijim granama su dodeljene uloge i procese sa ulogama. Kako je ovaj model dobijen proširivanjem X$d\pi$ računa ulogama, procesi, kao i u originalnom modelu, mogu međusobno komunicirati, menjati lokacije, koristiti i menjati podatke. Pored ovih mogućnosti, naši procesi mogu menjati i uloge na lokalnim podacima. Na taj način smo dobili model koji omogućava i aktivaciju i deaktivaciju uloga u lokalnim stablima podataka, tj. promenu prava pristupa. Jezik koji je prikazan u trećem poglavlju veoma je sličan ®X$d\pi$ računu. Postoji neznatna razlika u notaciji, kao i u tome što jezik u trećem poglavlju omogućava replikaciju proizvoljnog procesa i što umesto redukcionih konteksta koristi relaciju redukcije, zadatu pomoću relacije lokalne interakcije.

Dajemo definiciju mreže sa dobrim ponašanjem koja zahteva poštovanje zadatih politika lokacija, kao i da procesi ne prelaze ovlašćenja svojih uloga prilikom slanja, pristupa i promene podataka i da ne narušavaju konzistentnost podataka. Naime, na bilo kojoj lokaciji mreže sa dobrim ponašanjem:

- stablo sa podacima ne može da sadrži uloge koje su niže od minimalnih uloga propisanih od strane politike lokacije, a proces mora da ima bar jednu od tih minimalnih uloga;
- proces sa ulogama može da doda ulogu na granu lokalnog stabla samo ako je to dozvoljeno politikom lokacije;
- proces sa ulogama može da izbriše ulogu sa grane lokalnog stabla samo ako je to dozvoljeno politikom lokacije;
- dostupnost podataka ne sme biti prekinuta, tj. ukoliko proces može da pristupi nekoj grani, takođe mora moći da pristupi i njenoj roditeljskoj grani;
- grana stabla podataka nikada nije potpuno nedostupna;
- proces sa ulogama može da prenosi samo one vrednosti kojima može i sam da pristupi;
- proces sa ulogama pretražuje samo putanje kojima može da pristupi;
- proces sa ulogama može da pročita samo one podatke kojima može da pristupi;
- proces sa ulogama može da izbrše podstablo ukoliko može da pristupi svim granama tog podstabla.

Predlažemo tipski sistem i pokazujemo njegovu pouzdanost, tj. da se dobra tipiziranost mreže zadržava tokom njenog rada i da su mreže koje su dobro tipizirane, ujedno i mreže sa dobrim ponašanjem.

Četvrto poglavlje se odnosi na problem rukovođenja i kontrole pristupa podacima u distribuiranoj mreži sa podacima u RDF formatu, sa naglaskom na pitanja *privatnosti* posmatranih podataka.

Povećanjem međusobne povezanosti različitih izvora informacija, povećava se i njihova vrednost. Sa tom idejom, pre desetak godina, pokrenuta je inicijativa, nazvana Semantic Web za utvrđivanje odgovarajućeg opšteg formata za povezivanje podataka. Semantic Web se razvio u kolekciju preporuka za objavljivanje podataka na vebu [15, 14]. Očekuje se da veb povezanih podataka poraste u ogromni graf povezanih podataka, zasnovan na principima koje je formulisao Berners-Lee [11]:

- korišćenje URIs (IRIs) za imenovanje svega,
- korišćenje HTTP URIs (IRIs) da bi se ta imena mogla naći,
- korišćenje standardizovanih formata za objavljivanje podataka da bi se mogle dobiti funkcionalne informacije,
- mogućnost povezivanja podataka da bi se stvorila što bogatija mreža podataka.

Preporučeni standardi za objavljivanje i pretraživanje povezanih podataka su *Resource Description Framework (RDF)* [101, 131] i *SPARQL* [122]. RDF je okvir za predstavljanje informacija o njhovim izvorima. Bilo šta se može smatrati izvorom informacija, uključujući dokumente, ljude, fizičke objekte i apstraktne koncepte. Suštinu RDF-a predstavljaju *trojke* oblika

$$< \texttt{subjekat} > \ < \texttt{predikat} > \ < \texttt{objekat} >,$$

pomoću kojih se mogu praviti iskazi o izvorima informacija. RDF trojka izražava vezu između subjekta i objekta, pri čemu je priroda te veze izražena predikatom. U najopštijem obliku, podaci u RDF formatu su skupovi trojki. Elementi trojki mogu biti URIs (IRIs), literali i prazni čvorovi. Preporučeni jezik za pretraživanje RDF podataka je SPARQL, dok se SPARQL Update preporučuje kao standard za umetanje, brisanje i ažuriranje.

Kao što smo već napomenuli u [151], privatnost je definisana u smislu kontrole pristupa i prenosa podataka, tj. kao „mogućnost kontrolisanja ko ima pristup informacijama i kome se te informacije prenose". Na osnovu toga smatramo da je privatnost podataka zaštićena ako:

- vlasnik podataka uvek može da pristupi svim svojim podacima, i
- vlasnik podataka može da kontroliše ko ima pristup njegovim podacima, i
- vlasnik podataka može da vrši izmene nad svojim podacima.

Osim toga, privatnost se, pored na status nekih podataka, može odnositi i na relevantnost podataka za neku grupu ili na sposobnost čitalaca da razumeju podatke [134].

U ovom poglavlju razmatramo mrežu korisnika koji imaju svoje profile (podatke) u RDF formatu i procese koji se izvršavaju u njihovo ime. Da bismo omogućili da korisnici mogu da kontrolišu privatnost svojih podataka, imenima korisnika, resursa i trojki podataka dodeljujemo *politike zaštite privatnosti*. Slično kao u [127, 128], uzimamo SPARQL šablone kao politike zaštite privatnosti i kažemo da neki korisnik može pristupiti trojci podataka ako podaci koji odgovaraju tom korisniku zadovoljavaju `ASK` upitnik sačinjen na osnovu politike zaštite privatnosti te trojke. Ako je ime nekog korisnika dostupno nekom drugom korisniku, onda se i taj drugi korisnik takođe smatra vlasnikom podataka i ima pravo da ih menja. Povreda privatnosti bi bila izmena podataka nakon koje oni postaju nedostupni svom originalnom vlasniku i može biti prouzrokovana nekontrolisanim čitanjem podataka. Problem je sličan problemu iz trećeg poglavlja, te je stoga i za njegovo rešavanje pored metoda za kontrolu pristupa, potrebno uvesti dodatne metode za analizu podataka i procesa.

Postoji više radova na temu formalizacije različitih aspekata povezanih podataka. Procesni računi koji se pojavljuju u [81, 82] predstavljaju apstraktnu sintaksu za RDF i SPARQL pomoću koje se opisuju struktura povezanih podataka i upitnici. Autori radova [127, 128] opisuju model formalne semantike ontologije vezane za privatnost i predstavljaju aplikaciju koja omogućava korisnicima da kreiraju svoje politike privatnosti i kontrolišu pristup svojim podacima na osnovu

profila (podataka) onih korisnika koji upućuju zathev za čitanje tih podataka. U [42] autori proučavaju poreklo povezanih podataka pomoću procesnog računa, a tipski sistem statički određuje prava pristupa na osnovu porekla podataka. U ovom poglavlju, a i na sličan način u

- Svetlana Jakšić, Jovanka Pantović, Silvia Ghilezan: Linked Data Privacy, in Mathematical Structures in Computer Science, Cambridge University Press, FirstView:1-21, 2015. ([90]),

fokusiramo se na privatnost povezanih podatka. Uvodimo procesni račun koji predstavlja jezgro jezika procesa koji stupaju u interakciju sa podacima u RDF formatu. Ovaj procesni račun sadrži neke od operanada π-računa i modelira fragment jezika SPARQL , relavantan za analizirana pitanja privatnosti, kao i deo jezika višeg reda pomoću kojeg se RDF podaci konzumiraju. Podržane operacije su operacije čitanja, pisanja, brisanja i menjanja podataka. Politike privatnosti trojki se takođe mogu ažurirati. Procesi su zajedno sa podacima smešteni na korisnička imena koja u paralelnoj kompoziciji predstavljaju mrežu korisnika. Relacija redukcije data je preko relacije interakcije korisnika sa podacima. U relaciji redukcije se razlikuju slučajevi interakcije sa sopstvenim i sa tuđim podacima.

Na osnovu pokazanih svojstava zadovoljivosti politika korisnici čiji su podaci u potpunosti obrisani smatraju se blokiranim. Razlikujemo aktivne i blokirane korisnike. Definišemo odsustvo povreda privatnosti u definiciji mreža sa dobrim ponašanjem. Naime, privatnost u posmatranoj mreži smatramo zaštićenom ako:

- podaci aktivnog korisnika zadovoljavaju politiku imena tog korisnika i posredno, taj korisnik može da pristupi svim svojim podacima;
- korisnik čiji podaci zadovoljavaju politiku imena nekog drugog korisnika, ima omogućen pristup i svim podacima tog drugog korisnika;
- blokirani korisnici nemaju pravo da preuzimaju nove podatke iz mreže;
- se blokiranom korisniku ne mogu dopisati podaci;
- aktivni korisnici mogu da menjaju, ažuriraju i brišu podatke onih korisnika čija imena su im dostupna;
- blokirani korisnik može da dovrši akcije promena u mreži nad onim korisnicima čije su politike privatnosti niže od njegovih.

Predlažemo tipski sistem i pokazujemo njegovu pouzdanost, tj. da se, kao i u trećem poglavlju, dobra tipiziranost mreže zadržava tokom njenog rada i da su mreže koje su dobro tipizirane, ujedno i mreže sa dobrim ponašanjem.

Ključ jednostavnosti i efikasnosti ovog tipskog sistema leži u uvedenoj relaciji poređenja politika. Ono što razlikuje račun i tipski sistem predložene ovom poglavlju od drugih koji analiziraju mnoštvo osobina vezanih za bezbednosti i kontrolu pristupa, jeste što se ovde ne koriste dodatna sredstva za kontrolu privatnosti, osim onih koja su već prisutna u samom jeziku. Naime, politike zaštite privatnosti podataka izražene kao ASK upitnici, te se provera njihove zadovoljivosti svodi na zadovoljivost upitnika podacima korisnika koji pokušavaju da pristupe tim podacima.

Peto poglavlje se odnosi na problem prevencije *grešaka u memoriji* i *curenja memorije*, kao i *grešaka u komunikaciji* u generisanih programima napisanim na jeziku koji podržava razmenu poruka bez kopiranja, a u prisutvu izuzetaka.

Operativni sistem Singularity [84, 83] je primer sistema koji se snažno oslanja na paradigmu razmene poruka bez kopiranja. Procesi u ovom operativnom sistemu imaju pristup zajedničkom regionu koji se naziva *hip za razmenu*, komunikacija među procesima se odvija samo prenosom poruka preko kanala alociranih na hipu, a i same poruke su pokazivači na hipu. U [84, 83, 50] je detaljno objašnjeno da automatsko sakupljanje smeća nije praktično, te stoga to mora biti urađeno od strane procesa.

Razmena poruka bez kopiranja ima očiglednih prednosti u odnosu na konvencionalnije forme prenosa poruka. U isto vreme, može izazvati pojavu suptilnih grešaka u programiranju koje potiču od eksplicitnog rukovanja objektima i deljenja podataka. Zbog toga su i sami dizajneri operativnog sistema Singularity opremili Sing#, programski jezik na kome je napisan Singularity, eksplicitnim konstruktima (tipovima) i tehnikom za statičku analizu koja pomaže programerima da napišu kod u kojem nema nekoliko vrsta grešaka, uključujući: *greške u memoriji*, tj. pristup nealociranim ili dealociranim objektima na hipu; *curenja memorije*, tj. nagomilavanja nedosupnih alociranih objekata na hipu; *grešaka u komunikaciji* koje mogu prouzrokovati prekidanje procesa i biti okidač za prethodne vrste grešaka.

Neki aspekti Sing#-a su već bili formalizovani i proučeni u [44, 136, 146, 17]. Naime, kako Sing# koristi *ugovore kanala* za prevenciju grešaka u komunikaciji, u [17] je pokazano da se oni mogu pogodno predstaviti kao varijanta tipova sesija, i da se informacije dobijene od tipova sesija, zajedno sa linearnom disciplinom, mogu iskoristiti za prevenciju grešaka u memoriji i komunikaciji. U ovom poglavlju i u

- Svetlana Jakšić and Luca Padovani: Exception Handling for Copyless Messaging, in Science of Computer Programming, Vol. 84, pages 22-51, ISSN 0167-6423, Elsevier, 2014. ([89]), i

- Svetlana Jakšić, Luca Padovani: Exception Handling for Copyless Messaging in Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming PPDP 2012, pages 151-162, ACM, 2012. ([88]).

se fokusiramo na izuzetke i rukovanje njima. Dva glavna problema, prouzrokovana izuzecima, koja smo primetili kod Singularity operativnog sistema su:

- kako se, korišćenjem ugovora kanala, greške u komunikaciji sprečavaju komplementarnim akcijama procesa koji poseduju različite krajeve istog kanala, skok u toku izvršavanja jednog procesa, izazvan na primer izbacivanjem izuzetka, može da naruši ravnanje krajeva kanala i poremeti naredne interakcije, i

- kada se izbaci izuzetak, poruke koje su poslate a još nisu primljene, kao i objekti koji su alocirani od početka izvršavanja `try` bloka, mogu da postanu nedostupni, te da na taj način dođe do curenja memorije.

U Sing#-u postoje mehanizmi za rešavanje ovih problema, za koje pokazujemo da su limitirani i nezadovoljavajući.

Predlažemo rešenje koje kombinuje statičku analizu (inspirisanu postojećim radovima na temu rukovanja izuzecima u sesijama [25, 23, 24]) sa sve-ili-ništa sementikom `try` blokova, u formi transakcija. Osnovna ideja je da se `try` blok ili izvršava u potpunosti i da se tada njegovi efekti na hip trajno zabeleže ili da se prekida izbacivanje izuzetka. Ukoliko se to dogodi, svi procesi koji su bili ušli u transakciju se obaveštavaju o izbacivanju izuzetka da bi se tipovi krajeva kanala mogli ponovo poravnati, a hip se vraća na stanje u kom je bio pre početka `try` bloka. Između ostalog, da bi se izbeglo curenje memorije, neophodno je dinamički pratiti memoriju koja je alocirana tokom izvršavanja `try` bloka kako bi se ta memorija mogla pravilno vratiti u početno stanje u slučaju izbacivanja izuzetka.

Predlažemo procesni račun u kojem procesi mogu da primaju i šalju poruke bez kopiranja. Taj račun je, u suštini, varijanta π-računa, osim što imena predstavljaju pokazivače hip lokacije, umesto komunikacionih kanala. Dodatni operandi su slični odgovarajućim iz Sing#-a. Semantika operacija data je pomoću relacije redukcije koja ujedno opisuje izvršavanje procesa i njihove uticaje na hip.

Dajemo definiciju procesa sa dobrim ponašanjem koja zahteva odsustvo nedostupnih alociranih pokazivača i da nemogućnost daljeg izvršavanja procesa znači da je u jednom od stanja koja definišemo kao zaglavljena (uključujući, na primer i uzajamno blokiranje), a koja ne mogu proizvesti curenje memorije. Predlažemo tipski sistem koji omogućava prevenciju pomenutih grešaka, čak i u slučaju izbacivanja izuzetka, ali samo ukoliko su i odgovarajući delovi koda za rukovođenje tim izuzetkom prisutni. Neki procesi zapravo nikada neće prouzrokovati greške tokom izvršavanja iako se ne mogu tipizirati. U ovom poglavlju i u [89] smo tipski sistem iz [88] proširili relacijom podtipiziranja i na taj način značajno povećali preciznost tipskog sistema. Pokazujemo pouzdanost tog tipskog sistema sa podtipiziranjem, tj. da se dobra tipiziranost konfiguracija, sačinjenih od hipa i procesa, održava tokom izvršavanja procesa i da su dobro tipizirani procesi ujedno i procesi sa dobrim ponašanjem.

Jedna od ključnih ideja koje smo koristili je da u `try` blokovima „zaključamo" tipove pokazivača na krajeve kanala koji nisu odgovarajuće označeni (tj. nisu označeni kao tipovi vezani za transakciju) i zabranimo procesima da ih koriste. Na taj način tipski sistem može statički da obezbedi da procesi ne vrše modifikacije izvan dela hipa koji se lako može vratiti u stanje pre početka transakcije.

Koristimo invarijante garantovane tipskim sistemom da pokažemo kako se troškovi implementacije izuzetaka mogu smanjiti: redovi u kojima se nalaze poruke koje čekaju da budu primljene od strane nekog kraja kanala koji je ušao u transakciju su sigurno prazni na početku transakcije, te stoga vraćanje hipa na početno stanje znači, jednostavno, pražnjenje tog reda. Takođe, samo krajevi kanala koji su alocirani u toku transakcije mogu biti dealocirani u toku transakcije, te stoga vraćanje hipa na početno stanje ne iziskuje realociranje.

Šesto poglavlje sadrži zaključak teze i razmatra detalje aktuelnog i budućeg istraživanja kandidata.

Abstract

This thesis investigates problems of secure and private data access and administration in distributed networks and prevention of memory errors and leaks as well as the communication errors in copyless messaging communication paradigm. All problems are formalized in process calculi and type systems are used for verification of desired properties of systems and programs.

The outline of the thesis is the following. The thesis consists of six chapters, starting with the introductory and ending with the concluding chapter.

Brief overview of theoretical background is given in the second chapter. This chapter contains the basic notions of process calculi and type systems, together with the definition of π-calculus.

Each one of the next three chapters considers one of the problems investigated in the thesis. These chapters may be observed as separate entities which rely on the notions and calculus given in the second chapter. In particular, the third chapter addresses the problem of secure data administration in a distributed network containing XML data. We introduce a role-based access control calculus for modelling such networks. We define favourable security properties and use a type system to ensure that security of the network is not violated during the execution of processes.

The problem of privacy in a distributed network with data in RDF is investigated in the fourth chapter. We introduce a calculus to model such network and assign privacy protection policies to the users. We define favourable privacy properties in terms of access rights and use a type system to prove that these properties are preserved during the computation. The key of simplicity and effectiveness of the type system lies in the introduced policy comparison relation.

The fifth chapter deals with the prevention of memory errors and leaks as well as the communication errors in copyless messaging communication paradigm, in presence of exceptions. Message passing is a flexible paradigm that allows autonomous entities to exchange information and to synchronize with each other. While copying messages is inevitable in a distributed setting, the availability of a shared address space makes it possible to implement a copyless form of message passing, whereby only pointers to messages are exchanged. We formalize a core language of processes that communicate and synchronize through the copyless message passing paradigm and can throw exceptions. We study a type system guaranteeing some safety properties, in particular that well-typed processes are free from communication errors and do not leak memory even in presence of (caught) exceptions.

Contents

Chapter 1

Introduction

Today computer systems consist of multiple computers in a network, multiple applications run on a computer, multiple processors (or multi-core processors) operate on a computer, web sites have multiple users, and computations are often split in order to run faster. Therefore we must conclude that *concurrency* is present everywhere. *Communication* and *interaction* have become a central aspect of all modern software systems, which range from distributed processes connected by wide area networks down to collections of threads running on different cores within the same processing unit. The design of efficient methods for concurrency control in systems such as operating systems and database management systems that are generally assumed to operate indefinitely and without unexpected termination is a challenging task. It is difficult to detect bugs (race conditions, deadlocks, security violations,...) in the presence of concurrency because computations interleave nondeterministically. The diversity and complexity of problems related to concurrency has resulted in variety of *formal models*. In general, a formal model must be simple, expressive, defined by using techniques from logics and mathematics and it must provide a technique for analysis of a specific property. Some well-established models for concurrency are the actor model [74, 4], transactional memories [91, 66, 67], Petri nets [116, 124] and process calculi [105, 108, 75].

Three problems will be investigated in this thesis. The first one is data management and access control in a distributed network with semi-structured data in XML format, with the accent on the *security* of the observed data. Access control methods prevent unauthorized access to the data, but that solely is not enough to protect the security of information. In particular, the access is either allowed or denied, so there are no means to control how the data is used afterwards or if the data was influenced in an improper way. In order to prevent security violations in a network, additional methods for data management control must be introduced. Concurrency and distribution often present challenges for such methods.

The second problem is data management and access control in a distributed network with data in RDF format, with the accent on the *privacy* of the observed data. There is often the need to publish data in a format which allows structured and semi-structured data to be mixed. Such web of data is referred to as Web of Linked Data and it uses RDF to interlink, mix, expose and share data across different applications. A great merit of the Web of Linked Data is its exposure to public consumption. Even though public availability brings a great advantage

to users of such data, not all data are produced for public usage. For example, RDF is often used to represent personal information and data from social networks. This gives rise to the question of privacy of linked data, since the lack of privacy protection mechanisms often discourages people from publishing data on the Web of Linked Data. Addressing this issue requires a clear explanation for the intuition of the notion of privacy. In [151], the privacy is defined as "the ability to control who has access to information and to whom that information is communicated". Besides access control, as for the first problem, additional means are required in order to preserve privacy in a network.

The third problem is prevention of *memory errors* and *leaks* as well as *communication errors* in copyless messaging communication paradigm in presence of exceptions. Message passing is a flexible paradigm that allows autonomous entities to exchange information and to synchronize with each other. The term "message passing" seems to suggest a paradigm where messages move from one entity to another, although more often messages are copied during communication. While this is inevitable in a distributed setting, the availability of a shared address space makes it possible to implement a copyless form of message passing, whereby only pointers to messages are exchanged. Copyless messaging and exceptions are clearly at odds with each other: on the one hand, copyless messaging requires a very disciplined and controlled access to memory; on the other hand, exceptions are in general unpredictable and disrupt the normal control flow of programs. Prevention of aforementioned errors in presence of exceptions requires static analysis techniques and some support from the runtime system.

The objectives of the thesis are to formalize these problems and use type systems to verify certain properties. Process calculi focus on communication, interaction and synchronization, and as such are appropriate formal model for the problems considered in this thesis. The use of type systems is a prominent static method for verification and enforcement of variety of properties. In Chapter 2, we will give more details on the formalisms used in the thesis.

1.1 Publications and structure of the thesis

Chapter 2 contains basic notions of process calculi and type systems, together with the definition of π-calculus. Each of Chapters 3, 4 and 5 consider one of the problems investigated in the thesis. These chapters may be viewed as separate entities which rely on notions and calculus given in Chapter 2. Chapter 3 is based on the paper:

1. Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović: Types for Role-based Access Control of Dynamic Web Data in Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010), LNCS, Vol. 6559, pages 1-29, Springer, 2011. ([40]).

It introduces a role-based access control calculus for modelling dynamic web data and a corresponding type system. It is an extension of the X$d\pi$ calculus proposed

in [55]. A network is a parallel composition of locations, where each location contains processes with roles and a data tree whose edges are associated with roles. Processes can communicate, migrate from a location to another, use the data, change the data and the roles in the local tree. We obtain a model which controls access to data. We propose a type system which ensures that a specified network policy is respected during computations and which enforces desired security properties.

The foundation of Chapter 4 is the paper:

1. Svetlana Jakšić, Jovanka Pantović, Silvia Ghilezan: Linked Data Privacy, in Mathematical Structures in Computer Science, Cambridge University Press, FirstView:1-21, 2015. ([90]).

It introduces a calculus and a type system for privacy (access) control of RDF data. The introduced calculus presents a core language of processes that interact with data in RDF format. These processes together with data are enclosed under named users which are put in parallel, representing a network of users interacting with each other. We define the desired privacy properties of the network by defining well-behaved network. We have studied a type system guaranteeing some privacy properties by proving that well-typed network is well behaved. The key of the type system simplicity and effectiveness lies in the policy comparison relation. By defining a type system and proving its soundness, we have been able to verify preservation of privacy properties from [127, 128], where the vocabulary for fine-grained privacy preference control and a tool for privacy preference management were defined, as well as several more general properties.

Chapter 5 is based on the papers:

1. Svetlana Jakšić and Luca Padovani: Exception Handling for Copyless Messaging, in Science of Computer Programming, Vol. 84, pages 22-51, ISSN 0167-6423, Elsevier, 2014. ([89]);

2. Svetlana Jakšić, Luca Padovani: Exception Handling for Copyless Messaging in Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming PPDP 2012, pages 151-162, ACM, 2012. ([88]).

We introduce a calculus and a type system for copyless messaging that is able to guarantee the absence of communication errors, memory faults, and memory leaks in presence of exceptions. We have formalized a core language of processes that communicate and synchronize through the copyless message passing paradigm and can throw exceptions. In this context, where the sharing of data and explicit memory allocation require controlled policies on the ownership of heap-allocated objects, special care must be taken when exceptions are thrown to prevent communication errors (arising from misaligned states of channel endpoints) and memory leaks (resulting from messages forgotten in endpoint queues). We have studied a type system guaranteeing some safety properties, in particular that well-typed processes are free from communication errors and do no leak memory even in presence of (caught) exceptions. We have taken advantage of invariants guaranteed by

the type system for taming the implementation costs of exception handling: the queues of endpoints involved in a transaction are guaranteed to be empty when the transaction starts, so that state restoration in case of exception simply means emptying such queues; also, only endpoints local to a transaction can be freed inside the transaction, so that state restoration in case of exception does not involve re-allocations.

Chapter 6 concludes the thesis and discusses current and future work of the candidate.

Chapter 2

Background

Overview This chapter contains basic notions of process calculi and type systems. Section 2.1.1 describes the syntax and semantics of π-calculus.

2.1 Process calculi

The basis of concurrent computation is the notion of a *process*. The process concept and concurrency are the heart of modern operating [132] and distributed systems [139]. A process is an executing instance of a computer program. It may be able to perform more that one task at the time using concurrent units of execution named *threads*. Processes interact with each other by sending and receiving messages over communication mediums called *channels*.

2.1.1 π-calculus

The π-calculus [108] is a formal model for concurrent computation introduced by Milner, Parrow and Walker in early 90's. A π-calculus process is an abstraction of an autonomous entity willing to interact with other processes and a π-calculus channel is an abstraction of a communication link between two processes. The π-calculus is able to describe concurrent systems that may change configuration during computation by communication of channels along channels (channel transmission). By its syntax it is possible to represent the essence of concurrent systems: processes, parallel composition of processes, communication between processes, channel transmission, creation of fresh channels, nondeterminism and replication of processes. In the following paragraphs we give a version of the π-calculus which will serve as the base for process calculi presented in Chapters 3, 4 and 5.

We assume that we are given an infinite set $\mathsf{Channels}$ of *channel names* ranged over by $a, b, \ldots$, and an infinite set of $\mathsf{Variables}$ of *variables* ranged over by $x, y, \ldots$. Channel names and variables are together referred to as *names*. We let $u, v, \ldots$ range over elements of $\mathsf{Channels} \cup \mathsf{Variables}$.

The π-calculus processes are defined by the grammar given in Figure 2.1. The term 0 denotes the idle process that performs no action. Names are the only messages that are communicated trough channels. More precisely, the term $u?x.P$ denotes a process that first receives a name v on name u and then behaves as P

P	::=		**Process**
		0	inaction
	$\mid$	$u?x.P$	input
	$\mid$	$u!u.P$	output
	$\mid$	$P \mid P$	parallel
	$\mid$	$P \oplus P$	choice
	$\mid$	$(\nu a)P$	restriction
	$\mid$	$*P$	replication

Figure 2.1: Syntax of π-calculus processes.

$$
\begin{array}{rlrl}
\mathsf{fn}(0) &= \emptyset & \mathsf{bn}(0) &= \emptyset \\
\mathsf{fn}(u?x.P) &= \{u\} \cup (\mathsf{fn}(P) \setminus \{x\}) & \mathsf{bn}(u?x.P) &= \{x\} \cup \mathsf{bn}(P) \\
\mathsf{fn}(u!v.P) &= \{u, v\} \cup \mathsf{fn}(P) & \mathsf{bn}(u!v.P) &= \mathsf{bn}(P) \\
\mathsf{fn}(P \mid Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) & \mathsf{bn}(P \mid Q) &= \mathsf{bn}(P) \cup \mathsf{bn}(Q) \\
\mathsf{fn}(P \oplus Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) & \mathsf{bn}(P \oplus Q) &= \mathsf{bn}(P) \cup \mathsf{bn}(Q) \\
\mathsf{fn}((\nu a)P) &= \mathsf{fn}(P) \setminus \{a\} & \mathsf{bn}((\nu a)P) &= \{a\} \cup \mathsf{bn}(P) \\
\mathsf{fn}(*P) &= \mathsf{fn}(P) & \mathsf{bn}(*P) &= \mathsf{bn}(P)
\end{array}
$$

Figure 2.2: Free and bound names of processes.

where x has been replaced by v, while the term $u!v.P$ denotes a process that sends name v on name u and then behaves as P. The term $P \mid Q$ denotes a parallel composition of processes P and Q. It can execute all sequences of messages sent and receive by both P and Q interleaved in any order. If P and Q share a channel and one is willing to send a message while the other one is willing to receive the message, then $P \mid Q$ can perform an internal communication. The term $P \oplus Q$ denotes a process that nondeterministically decides to behave as either P or Q. The restriction (νa) in the term $(\nu a)P$ denotes that the channel a is not used outside of process P, i.e. is fresh in process P. The term $*P$ denotes a process that can behave as P an infinite number of times.

We adopt some standard conventions regarding the syntax of processes: we sometimes use a prefix form for parallel compositions and write, for example, $\prod_{i=1..n} P_i$ instead of $P_1 \mid \cdots \mid P_n$; we identify $\prod_{i \in \emptyset} P_i$ with 0; we omit trailing occurrences of 0. We give the following order of operator precedence from highest to lowest: input and output prefixes, replication, parallel composition and choice.

The name binders of the calculus are input prefix and restriction. Free and bound names of a process P, denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are defined in Figure 2.2. Input prefix is the only binder for the variables. Free and bound variables of a process P, denoted by $\mathsf{fv}(P)$ and $\mathsf{bv}(P)$, are defined in Figure 2.3. We say that process P is a *closed process* if it has no free variables. We often use process to refer to closed processes.

The notion of computation in a process calculus can be given by a *reduction relation* or a by *labelled transition system*. We give the operational semantics for the present variant of the π-calculus in terms of a *structural congruence* over processes and a reduction relation. Structural congruence identifies structurally

$$
\begin{array}{rlrl}
\mathsf{fv}(0) &= \emptyset & \mathsf{bv}(0) &= \emptyset \\
\mathsf{fv}(u?x.P) &= (\{u\} \cap \mathsf{Variables}) \cup (\mathsf{fv}(P) \setminus \{x\}) & \mathsf{bv}(u?x.P) &= \{x\} \cup \mathsf{bv}(P) \\
\mathsf{fv}(u!v.P) &= (\{u, v\} \cap \mathsf{Variables}) \cup \mathsf{fv}(P) & \mathsf{bv}(u!v.P) &= \mathsf{bv}(P) \\
\mathsf{fv}(P \mid Q) &= \mathsf{fv}(P) \cup \mathsf{fv}(Q) & \mathsf{bv}(P \mid Q) &= \mathsf{bv}(P) \cup \mathsf{bv}(Q) \\
\mathsf{fv}(P \oplus Q) &= \mathsf{fv}(P) \cup \mathsf{fv}(Q) & \mathsf{bv}(P \oplus Q) &= \mathsf{bv}(P) \cup \mathsf{bn}(Q) \\
\mathsf{fv}((\nu a)P) &= \mathsf{fv}(P) & \mathsf{bv}((\nu a)P) &= \mathsf{bv}(P) \\
\mathsf{fv}(*P) &= \mathsf{fv}(P) & \mathsf{bv}(*P) &= \mathsf{bv}(P)
\end{array}
$$

Figure 2.3: Free and bound variables of processes.

[S-Par Idle] $P \mid 0 \equiv P$

[S-Par Commutativity] $P \mid Q \equiv Q \mid P$

[S-Par Associativity] $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

[S-Replication] $*P \equiv P \mid *P$

[S-Res Idle] $(\nu a)0 \equiv 0$

[S-Res Swap] $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$

[S-Res Scope] $a \notin \mathsf{fn}(P) \Rightarrow (\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$

Figure 2.4: Structural congruence

equivalent processes. It is the smallest equivalence relation that includes alpha conversion (renaming of bound names) and rules given in Figure 2.4. Namely, rules [S-Par Idle], [S-Par Commutativity] and [S-Par Associativity] are commutative monoid laws for parallel composition. Rule [S-Replication] states that the replication of a process can be considered as infinite number of copies of that process running in parallel. Rule [S-Res Idle] states that since there are no names in the idle process, the restriction in front of it can be removed. The names can be restricted in an arbitrary order, as stated in rule [S-Res Swap]. The most interesting rule is [S-Res Scope], often called "scope extrusion", which states that the scope of a restricted name can be extended to a process which does not have this name as a free name.

We say that process P *reduces to* Q if $P \to Q$ can be derived by an application of the *reduction rules* of Figure 2.5. In the rest of this paragraph we give informal descriptions of these rules. Rule [R-Communication] contains the primitive step of computation in π-calculus. The processes $a!b.P$, which sends name b on name a, and $a?x.Q$, which is willing to receive a name on name a, running in parallel reduce to processes P and Q running in parallel, where name b substitutes variable x in Q. Rule [R-Parallel] specifies that if process P is able to reduce, then the same reduction can happen if P is put in parallel with some process R. Similarly, rule [R-Restriction] enables reduction under restriction. Rule [R-Choice] states that a process $P \oplus Q$ nondeterministically reduces to either P or Q. The last rule, [R-Structural Congruence], states that process P can reduce to process Q whenever both processes can be rearranged by the structural congruence so that obtained processes are able to reduce. We use $\to^*$ to denote reflexive and transitive closure of $\to$.

$$\text{[R-Communication]}\quad a!b.P \mid a?x.Q \to P \mid Q\{b/x\}$$

$$\text{[R-Parallel]}\quad \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \text{[R-Restriction]}\quad \frac{P \to Q}{(\nu a)P \to (\nu a)Q} \qquad \text{[R-Choice]}\quad \frac{i \in \{1,2\}}{P_1 \oplus P_2 \to P_i}$$

$$\text{[R-Structural]}\quad \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

Figure 2.5: Reduction relation

More operators, such as standard summation (external choice), are going to be introduced and discussed later. It is possible to encode `if-else` commands with the non-deterministic process $P \oplus Q$ omitting the condition that determines the chosen branch. This is just one of many variations of the language. Its choice is induced by the need of having a base for process calculi presented in the thesis, that model more complex systems. The π-calculus given in this section will be slightly adapted and extended in order to fit the envisaged problems.

More detailed presentations of the variations, syntax, semantics, equivalences and axiomatisations one could find in [107, 114, 130]. Nowadays there is a large family of calculi for concurrent computing with the π-calculus as a core [72, 70, 28, 3, 51, 115, 79, 121, 123, 26, 53].

2.2 Type systems

Roots of type systems used in computer science reach to the beginning of the 20th century when *type theory* was introduced in works of Whitehead and Russell [152] as an alternative to set theory. Further well-known type theoretic contributions to the field of mathematical foundations are the works of Church [30] and Martin-Löf [102] and currently active Homotopy type theory [140].

Practical impact of type theory reflects in a variety of areas of mathematics, logics and computer science such as proof assistants [13, 111, 112, 138, 68, 110, 34, 118] and programming languages [119, 109] and even broadly in linguistics [97, 141] and social sciences [135, 7].

Type systems are one of the most widely used techniques for programming languages analysis. Type systems are used to avoid undesired behaviors (*run-time type errors*), which are specific for each language. A type system, in general, splits elements of the language, called *terms*, into sets, called *types*, and proves absence of certain errors on the basis of the types that are thus assigned. It may happen that some terms are rejected by the type system although they always behave correctly on run-time. The design of more precise type system enables more terms to be typed but at the cost of possibly more complex *typechecking algorithm*, so the balance between these two issues is important.

Type systems in concurrent programs and systems are extensively used to eliminate run-time errors such as deadlocks and race conditions as well as to analyse whether a process behaves in a disciplined manner, to extract information about programs that can be helpful to reason about their run-time behavior etc.

There are many type systems for π-calculus and other process calculi, starting from simple sorting and typing systems designed for problems such as arity mismatch and communication errors in case of polyadic π-calculus [106, 58, 145] to more specialized purposes such as deadlock freedom [93] and secure information flow [80]. Further examples include type disciplines that address consistent usage of names for input and output [120], impose linearity constraints on usage of names [94], present general type framework [85] etc. See [92] for more details on types for π-calculus.

Type systems in the thesis are related to specific type systems for π-calculus. Types introduced in [40] and [90] are related to *simple types* from Chapter 6 of [130]. Simple types prescribe that only data of the expected type are exchanged. Types introduced in [88, 89] are similar to *session types* of [144, 62]. The key idea of *session types* is to type channel names with sequences of types that prescribe the trace of the channel usage. They ensure the absence of communication errors and the conformance to the session protocol. The original session types from [78, 79] do not initially distinguish between two channel ends while in [144, 62] the authors define endpoint types. See [38] for a survey on sessions and session types.

In order to describe typed calculi we need to introduce some standard terminology and notation as in Chapter 6 of [130]. An expression of the form $u : T$, where u is a name and T is a type, *assigns* a type to a name. A *type environment* is a finite set of assignments of types to names. It is assumed that names of the type environment are all different. We use Γ to range over type environments. *Type judgements* are expressions of the form $\Gamma \vdash e : T$ where e is a term of the language. Sometimes, when there is no ambiguousness, T is omitted. A type judgement $\Gamma \vdash e : T$ indicates that term e has type T. A term of a language is *well typed in* Γ if there is T such that $\Gamma \vdash e : T$ is derivable from the axioms and the inference rules of the type system.

A type system is *sound* if well-typed terms cannot cause run-time errors during computation. Terms that do not cause run-time errors during computation are called *well behaved*. In order to prove that a type system is sound one should usually prove two main results: well-typedness is preserved during computation (*subject reduction*), and well-typed terms are well behaved (*type safety*). The formalization of these results depends on the language and the type system under consideration, and their proofs usually require a number of additional properties.

Chapter 3

Types for secure access

Overview This chapter is based on the paper "Types for Role-Based Access Control of Dynamic Web data" [40]. In Section 3.1, we describe the setting and the problem we are addressing. In Section 3.2, we introduce a role-based access control calculus for modelling dynamic web data, as an extension of X$d\pi$ calculus proposed in [55]. In Section 3.3, we define a type system which ensures that the specified security protection policies are respected during computations. In Section 3.4, we prove the soundness of the type system with respect to the definition of well-behaved networks given in Section 3.2.4.

3.1 Dynamic web data and secure access

Data present on the Internet are often in formats that conform to a flexible structure which depends on the purpose of the data. In such semi-structured data, there is no separation between data and schema, as there is in structured data, present in relational databases. Semi-structured data are accessible, portable and suitable for exchanges between databases of different kinds. *Extensible Markup Language* (XML) [19] and XQuery [16, 126] currently belong to the standard methods for expressing and querying semi-structured data.

XML is a language designed to store and transport data and is readable by both humans and machines. Documents written in XML have a tree structure that starts at *the root* and branches to *the leaves*. *Elements of the trees* begin with element's start tag, < `tag` >, and finish with element's end tag, < /`tag` >. Tags can have arbitrary names. Elements can contain text, attributes and other elements. The terms *parent*, *child*, *ancestor*, *descendant* and *sibling* are used to describe relationships between elements of the XML trees. Siblings are children on the same level. For example, in Figure 3.1, `root` element is the parent of two sibling `child` elements and the ancestor of two `subchild` elements.

Recommended language for querying XML data is XQuery which can be used to extract information for further use, generate reports, transform XML data and search web documents. It features XPath (a syntax for identifying parts of XML data [9]) expressions, as well as, functions and `for`, `let`, `where`, `order by`

```
< root >
  < child >
    < subchild > ··· < /subchild >
                 ⋮
  < /child >
  < child >
    < subchild > ··· < /subchild >
                 ⋮
  < /child >
< /root >
```

Figure 3.1: XML data structure.

and `return` expressions. There is an extension of the XQuery language which provides means of modifying XML documents or data, XQuery Update Facility [125]. It enables insertion, deletion, modification and creation of nodes.

One of the essential steps in managing distributed systems with data is the security administration, which is one of the main features in prevention of unauthorized access to system resources. Role-based access control (RBAC) [129] is an access control method that relies on the notions of users, roles and permissions. It controls the access of users to the system resources in accordance with the activities they have to perform in the system. In accessing the system resources, a user has those permissions which are assigned to its roles. In a system, roles are statically defined by the organisation structure, hence the security administration is reduced to the management of permissions. This makes RBAC a simple and desirable access control technology.

Security of data is defined in terms of preservation of confidentiality, integrity and availability. Confidentiality is about to whom the data are transmitted. Integrity refers to unauthorized modification of the data. It is often phrased as preservation of some important property, such as consistency, accuracy or meaningfulness of data. In this sense, we deem that the security of data is protected if users (processes) do not exceed rights assigned to their roles i.e.

- if the data is transmitted only to the parties which are allowed to access them, and
- if processes access data respecting predetermined policies, and
- if processes do not change the data in such way that the data becomes inconsistent with respect to predetermined policies.

Access control methods prevent unauthorized access to the data, but that solely is not enough to protect the security of information. In particular, the access is either allowed or denied, so there are no means to control how the data is used afterwards or if the data was influenced in an improper way. So, in order to preserve security, further methods for data and process analysis are needed.

```
< music >
  < song >
    < title > Yellow sun < /title >
                  ⋮
  < /song >
  < song >
    < title > Blue moon < /title >
                  ⋮
  < /song >
< /music >
```

Figure 3.2: Example of XML data.

Peer-to-peer networks with dynamic web data are modelled in X$d\pi$ calculus presented in [55]. The data model is a basic model of semi-structured data, unordered labelled trees with pointers for referring to other parts of the network with embedded processes for querying and updating such data. The authors chose edge-labelled rooted trees as model for XML documents. Essentially, it is a graph in which edges are labeled with element tags. Another choice, they made, was to embed processes and pointers only in leaves instead throughout the tree. Both of these choices, although slightly different from the standard model of XML data, did not influence the ideas presented in [55]. The paper [41] discusses a security type system for the X$d\pi$ calculus based on security levels. In [40] we focus on security properties based on RBAC. A brief summary of the research presented in [40] and in this chapter is:

- we formalize a calculus which applies RBAC to a network with semi-structured data;
- we develop a type system for preventing security violations and we show its soundness.

3.1.1 Examples and motivation

In Figure 3.2, we give an example of XML data. The root element's tag is `music`, which is the parent of two sibling `song` elements. These siblings have children with tags `title`. As in [55], we will use edge-labelled rooted trees which contain data only in the leaves as model of XML documents. In this example, leaves contain text. XQuery language uses the function `doc` to open files and XPath expressions to navigate through XML elements. If we assume that the data from Figure 3.2 are the contents of the file `"musicbox.xml"`, then the following query could be used to select all `title` elements in the this file:

```
doc("musicbox.xml")/music/song/title
```

It would extract the following data

```
< title > Yellow sun < /title >
< title > Blue moon < /title > .
```

```
< music >
  < song  category = "demo" >
    < title > Yellow sun < /title >
                  ⋮
  < /song >
  < song  category = "full" >
    < title > Blue moon < /title >
                  ⋮
  < /song >
< /music >
```

Figure 3.3: Example of XML data wtih attributes.

The query

```
for $x in doc("musicbox.xml")/music/song/title
return $x/title
```

returns the same XML data, while the query

```
for $x in doc("musicbox.xml")/music/song/title
return data($x)
```

returns only the data inside `title` element. It is possible to write queries which return data in HTML format.

Figure 3.3 shows the same data as Figure 3.2, except that some elements have attributes. In particular, two sibling `song` elements have two different values for `category` attribute. It could be expected that the song in `demo` category is accessible by everyone, contrary to the full versions of songs. In the context of RBAC, we could annotate the data so that we control the access in a desired way. Accessibility should not be discontinued, meaning that those that are allowed to access a child element must be allowed to access the parent element. Furthermore, we consider systems in which changes of data and access rights is possible. The processes that change data must be allowed to do so and reasonably have higher roles than those that just read data.

Now, let us consider a network of distinct locations, each containing XML data and processes. It can be expected that not all data should be accessible by all processes and that data may be changed only by authorized processes. For this reason, roles are assigned to both, data and processes. We suppose that each location has a *security protection policy* which prescribes which roles can access the location. Only processes that are allowed to access all of the data they aim to change, are allowed to do so. Since it should be possible to change access rights, this should, also, be done according to the location policy. A security violation would be any action which is not permitted or which can make the data inconsistent with respect to the location policy. A security violation may be caused by uncontrolled reading or transmitting of data.

3.2 Language

Notation Let us assume that there is an infinite set Channels of *channel names* annotated with *value types* ranged over by $a^{T_V}, b^{T_V}, \ldots$ and an infinite set Variables of *variables* ranged over by $x, y, \ldots$. We let symbols $u, v, \ldots$ range over elements of Channels $\cup$ Variables. Further on, we assume that there is a set Locations of *location names* ranged over by $l, m, \ldots$ and a set Edges of *edge labels* ranged over by $\mathsf{a}, \mathsf{b}, \ldots$ We let symbol λ range over elements of Locations $\cup$ Variables.

Roles We assume given a countable set Roles of *roles* ranged over by $r, s, \ldots$ Let $(\mathsf{Roles}, \sqsubseteq)$ be a lattice and let $\bot, \top \in \mathsf{Roles}$ be its bottom and top elements, respectively. The join operation is denoted by $\sqcup$. We use $\alpha, \beta, \ldots, \rho, \sigma, \ldots$ to denote non-empty sets of roles and $\tau, \zeta, \ldots$ to denote sets of roles containing $\top$. We use $\mathcal{E}, \mathcal{D}, \ldots$ to denote subsets of $\{(\rho, r) : \rho \subseteq \mathsf{Roles}, r \in \mathsf{Roles}\}$. We introduce two relations and two operations on the sets of roles.

Definition 3.2.1. *We say that set σ is* accessible *to the set ρ, and write $\sigma \leq \rho$, if there is one role in the set σ that is smaller than or equal to one role in ρ, i.e.*

$$\sigma \leq \rho \iff (\exists s \in \sigma)(\exists r \in \rho) s \sqsubseteq r.$$

We say that set ρ complies *with the set σ, and write $\sigma \leqq \rho$, if for every role in ρ there exist a smaller or equal role in σ, i.e.*

$$\sigma \leqq \rho \iff (\forall r \in \rho)(\exists s \in \sigma) s \sqsubseteq r$$

We disable *role r from the set of roles ρ, written $\rho \setminus\setminus r$, as follows:*

$$\rho \setminus\setminus r = \{s \in \rho | s \not\sqsubseteq r\}$$

The join operation for roles is extended to the sets of roles, as follows

$$\rho_1 \sqcup \rho_2 = \{r_1 \sqcup r_2 | r_1 \in \rho_1, r_2 \in \rho_2\}$$

3.2.1 Syntax

Data The data model is an unordered edge-labelled *data tree*, denoted by D, $E, \ldots$ The syntax of data trees is given by the grammar in Figure 3.4. The term $\emptyset_D$ denotes empty data tree. Data trees or subtrees may be variables. The term $D_1 \mid D_2$ denotes composition of data trees joining the roots. Every edge in a data tree is annotated with a set of roles that contains $\top$. The term $\mathsf{a}^{\tau}[\mathsf{V}]$ denotes data tree consisting of a tree edge with label a annotated with set of roles τ and *data term* V. Data trees, *scripts* and *pointers* belong to the syntax of data terms, as stated in Figure 3.4.

A script $\square\Pi$ is a static process embedded in a tree. It can be activated by a process from the same location. We use Π to range over processes with roles and variables.

A *path* p identifies data in a tree. The syntax of paths is given in Figure 3.4. A path is either an edge with a label annotated with a set of roles, a variable or a composition of two paths.

D	::=		**Data tree**	p	::=		**Path**
		$\emptyset_D$	empty data tree			$\mathtt{a}^\alpha$	edge
	|	x	variable		|	x	variable
	|	$D \mid D$	composition		|	p/p	composition
	|	$\mathtt{a}^\tau[\mathsf{V}]$	edge				
V	::=		**Data term**				
	|	D	data tree				
	|	$\Box\Pi$	script				
	|	$p@\lambda$	pointer				

Figure 3.4: Syntax of data trees, paths and data terms.

$$\chi ::= \Box x^{(\sigma,\mathcal{E},\mathcal{D})} \mid y^\alpha @ x^{(\sigma,\mathcal{E},\mathcal{D})} \mid x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$$

Figure 3.5: Syntax of patterns.

A pointer $p@\lambda$ refers to the data identified by the path p at location λ.

The syntax allows the leaves of a data tree to contain an empty data tree, a script or a pointer. In the examples, we also allow arbitrary texts in the leaves.

Patterns The syntax of *patterns* is given in Figure 3.5. They appear as arguments in commands for reading and changing data, where they are used, together with paths, to identify data terms in data trees. The term $\Box x^{(\sigma,\mathcal{E},\mathcal{D})}$ denotes a script pattern whose variable is decorated with three sets of roles. The term $y^\alpha @ x^{(\sigma,\mathcal{E},\mathcal{D})}$ denotes a pointer pattern whose path variable is decorated with a set of roles and whose location variable is decorated with three sets of roles. The term $x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$ denotes a data tree pattern whose variable is decorated with five sets of roles. In Section 3.3 we will relate all these sets of roles with the types of corresponding data terms. By $|\chi|$ we denote the data term obtained from χ by erasing all decorations.

Processes We distinguish two kinds of processes: *pure processes* and *processes with roles*. Their definitions, given in Figure 3.6, are mutually recursive.

Pure processes We can divide pure processes into four groups:

1. the terms $0, u?x.P, u!\mathsf{v}.P, P \mid P$ and $*P$ are π-calculus processes. Symbol v ranges over values defined in Figure 3.7.

2. the term $\texttt{GO}\ \lambda.R$ denotes a process that will migrate to location λ and there continue as R.

3. the terms $\texttt{RUN}_p$, $\texttt{READ}_p(\chi).P$ and $\texttt{CHANGE}_p(\chi, \mathsf{V}).P$ denote processes that manage the data.

   In particular, the term $\texttt{RUN}_p$ denotes a process that activates scripts identified by path p.

P	::=		**Process**	R	::=		**Process with roles**
		0	inaction			$P^{\neg\rho}$	single process
	$\mid$	$u?x.P$	input		$\mid$	$R \mid R$	parallel
	$\mid$	$u!\mathsf{v}$	output		$\mid$	$(\nu a^{T\mathsf{v}})R$	restriction
	$\mid$	$P \mid P$	parallel				
	$\mid$	$*P$	replication				
	$\mid$	$\texttt{GO}\ \lambda.R$	migrate				
	$\mid$	$\texttt{RUN}_p$	run scripts				
	$\mid$	$\texttt{READ}_p(\chi).P$	read data				
	$\mid$	$\texttt{CHANGE}_p(\chi, \mathsf{V}).P$	change data				
	$\mid$	$\texttt{ENABLE}_p(r).P$	enable role				
	$\mid$	$\texttt{DISABLE}_p(r).P$	disable role				

Figure 3.6: Syntax of processes.

$$\mathsf{v} ::= u \mid \Box R \mid \lambda \mid p \mid D$$

Figure 3.7: Syntax of values.

The term $\texttt{READ}_p(\chi).P$ denotes a process that reads data terms identified by path p and pattern χ and then continues as P. The pattern variables are substituted with the identified data terms in P.

The term $\texttt{CHANGE}_p(\chi, \mathsf{V}).P$ denotes a process that changes data identified by path p and pattern χ with V and then continues as P. Pattern variables may appear in V and before the change, they are substituted by the identified data terms.

4. the terms $\texttt{ENABLE}_p(r).P$ and $\texttt{DISABLE}_p(r).P$ denote processes that manage roles in the data.

   In particular, the term $\texttt{ENABLE}_p(r).P$ denotes a process that allows role r to access additional part of the data identified by the path p.

   The term $\texttt{DISABLE}_p(r).P$ denotes a process that forbids role r to access data beyond the data identified by the path p.

We use $P, Q, \ldots$ to range over pure processes.

Processes with roles The term $P^{\neg\rho}$ denotes pure process P with set of roles ρ assigned to it. The term $R_1 \mid R_2$ denotes the parallel composition of processes with roles. The term $(\nu a^{T\mathsf{v}})R$ denotes restriction of channel $a^{T\mathsf{v}}$ in process R, as in π-calculus. Processes with possibly different roles can share the same private channel.

Locations and networks The syntax of networks is given in Figure 3.8. The term $l[D \parallel R]$ denotes a location named l which encloses data tree D and process

N	::=		**Network**
		$l[D \parallel R]$	location
	|	$N \,[\!]\, N$	parallel
	|	$(\nu a^{T\vee})N$	restriction

Figure 3.8: Syntax of networks.

with roles R. The term $N_1 \,[\!]\, N_2$ denotes parallel composition of networks. We let restricted channels be shared between several locations. This is denoted by the term $(\nu a^{T\vee})N$.

Location policies We assume given a function $\mathcal{P}$ that assigns *security protection policies* to location names. Sometimes, we refer to security protection policies as location policies or just policies. A security policy is a triple $(\sigma, \mathcal{E}, \mathcal{D})$. The data *accessibility policy* is given by the set σ. It is the set of minimal roles a process is required to have to access the data at location protected by the policy. The *administration policy* is given by two other sets which prescribe roles that a process should have in order to be allowed to change access rights, i.e. to enable and disable roles on the edges of the local data tree. In order to have more flexible administration policy we define an extension of the set membership relation for sets $\mathcal{E}$ and $\mathcal{D}$.

Definition 3.2.2.

$$(\rho, r) \in^{+} \mathcal{E} \Longleftrightarrow \exists (\rho', r') \in \mathcal{E} \text{ such that } \rho' \leq \rho \text{ and } r' \sqsubseteq r;$$

$$(\rho, r) \in^{-} \mathcal{D} \Longleftrightarrow \exists (\rho', r') \in \mathcal{D} \text{ such that } \rho' \leq \rho \text{ and } r \sqsubseteq r'.$$

We have introduced $\in^{+}$ and $\in^{-}$ in order to:

- allow processes with higher roles to modify access rights which processes with lower roles can already modify (condition $\rho' \leq \rho$);
- allow to enable higher roles when lower roles can be enabled (condition $r' \sqsubseteq r$);
- allow to disable lower roles when higher roles can be disabled (condition $r \sqsubseteq r'$).

We say that a process with roles ρ can enable role r if $(\rho, r) \in^{+} \mathcal{E}$, and that a process with roles ρ can disable role r if $(\rho, r) \in^{-} \mathcal{D}$.

A location policy is *well formed* if it does not permit role $\top$ to be enabled or disabled by any role and if the roles involved in changing access right are only the roles of some edges in the local tree. Formally:

Definition 3.2.3 (Well-formed policy). *A location policy $(\sigma, \mathcal{E}, \mathcal{D})$ is well formed if $(\rho, r) \in \mathcal{E} \cup \mathcal{D}$ implies $r \neq \top$ and $\sigma \leqq \rho \cup \{r\}$.*

We consider only well-formed policies.

Syntactic conventions We adopt some standard conventions regarding the syntax of processes:

- we sometimes use a prefix form for parallel compositions and write, for example, $\prod_{i=1..n} R_i$ instead of $R_1 \mid \cdots \mid R_n$;
- we identify $\prod_{i\in\emptyset} R_i$ with 0;
- we sometimes use $\vec{\nu}$ to denote $(\nu a_1^{T\mathsf{v}_1}) \ldots (\nu a_n^{T\mathsf{v}_n})$;
- we omit trailing occurrences of 0 and write, for example, $\mathtt{ENABLE}_p(r)$ instead of $\mathtt{ENABLE}_p(r).0$;
- we sometimes omit curly braces when there is only one element in the set and write, for example, (ρ, r) instead of $\{(\rho, r)\}$ and r instead of $\{r\}$;

Example 3.2.4. The data from Figure 3.2 corresponds to the following data tree

$$\mathtt{music}^{\tau_1}[\mathtt{song}^{\tau_2}[\mathtt{title}^{\tau_3}[\mathtt{Yellow\ sun}]\ldots] \mid \mathtt{song}^{\tau_4}[\mathtt{title}^{\tau_5}[\mathtt{Blue\ moon}]\ldots]]$$

The first and the second query from Section 3.1.1 correspond to the following process

$$\mathtt{READ}_{\mathtt{music}^{\tau}/\mathtt{song}^{\tau'}}(\chi).$$

The third query, which returns data inside the title element, corresponds to

$$\mathtt{READ}_{\mathtt{music}^{\tau}/\mathtt{song}^{\tau'}/\mathtt{title}^{\tau''}}(\chi).$$

We left edge annotations and patterns unspecified. This example suggests that our paths identify data at their ends. Notice that paths of XPath and XQuery identify data slightly different (at the beginning of final edge), but this is purely a matter of choice. The exact way how paths identify data in our calculus will be explained in the next section.

△

Example 3.2.5. Let us assume that there are roles in the following order $\bot \sqsubseteq \mathtt{guest} \sqsubseteq \mathtt{member} \sqsubseteq \mathtt{owner} \sqsubseteq \top$ and $\bot \sqsubseteq \mathtt{administrator} \sqsubseteq \mathtt{owner} \sqsubseteq \top$. We also assume $\mathtt{guest} \not\sqsubseteq \mathtt{administrator}$. In this and in several other examples we will observe the following network

$$\mathtt{musicbox}[D_{\mathtt{mb}} \parallel R_{\mathtt{mb}}] \,[]\, \mathtt{repository}[D_{\mathtt{rp}} \parallel R_{\mathtt{rp}}].$$

Let us assume that the privacy policy of `musicbox` is as follows:

$$\mathcal{P}(\mathtt{musicbox}) = (\{\mathtt{guest}\}, (\{\mathtt{owner}\}, \mathtt{guest}), (\{\mathtt{owner}\}, \mathtt{member})).$$

This policy is well formed because $\mathtt{guest} \neq \top$ and $\mathtt{member} \neq \top$ and $\{\mathtt{guest}\} \leqq \{\mathtt{owner}\} \cup \{\mathtt{guest}\}$ and $\{\mathtt{guest}\} \leqq \{\mathtt{owner}\} \cup \{\mathtt{member}\}$. The accessibility policy is such that the lowest role which may be allowed to access some data is `guest`. The administration policy consists of two sets, each containing just one pair. Let

us denote them by $\mathcal{E}_{\texttt{mb}}$ and $\mathcal{D}_{\texttt{mb}}$. As an illustration of the relations $\in^+$ and $\in^-$, the following holds

$$(\{\top\}, \texttt{guest}) \in^+ \mathcal{E}_{\texttt{mb}}$$
$$(\{\texttt{owner}\}, \texttt{member}) \in^+ \mathcal{E}_{\texttt{mb}}$$
$$(\{\texttt{member}\}, \texttt{guest}) \notin^+ \mathcal{E}_{\texttt{mb}}$$
$$(\{\texttt{owner}\}, \bot) \notin^+ \mathcal{E}_{\texttt{mb}}$$
$$(\{\top\}, \bot) \in^- \mathcal{D}_{\texttt{mb}}$$
$$(\{\texttt{owner}\}, \texttt{owner}) \notin^- \mathcal{D}_{\texttt{mb}}.$$

Role `administrator` cannot be enabled nor disabled at location `musicbox` because

$$(\{\texttt{owner}\}, \texttt{administrator}) \notin^+ \mathcal{E}_{\texttt{mb}}$$
$$(\{\texttt{owner}\}, \texttt{administrator}) \notin^- \mathcal{D}_{\texttt{mb}}.$$

Furthermore, this policy will not allow a process with role `administrator` to run at location `musicbox` because `guest` $\not\sqsubseteq$ `administrator`. We leave specification of data and processes with roles for the further examples. △

3.2.2 Data accessibility and identification

In our language data, paths and processes are annotated with sets of roles, which control access to data. We now formalize such access control by introducing two notions: a permission to access data terms and identification of data terms.

Each process with at least one role bigger than or equal to some role in σ has the permission to *access* an edge $\texttt{a}^\sigma$, i.e. if set σ is accessible to the set of process's roles. More formally:

Definition 3.2.6 (Edge access). *The edge* $\texttt{a}^\sigma$ *is accessible to a process with roles* ρ *if* $\sigma \leq \rho$.

We extend this definition to paths and say that a process with roles ρ can access path $\texttt{a}_1^{\alpha_1}/\ldots/\texttt{a}_n^{\alpha_n}$ if $\alpha_n \leq \rho$. Although it would be expected that a process with roles ρ can access path $\texttt{a}_1^{\alpha_1}/\ldots/\texttt{a}_n^{\alpha_n}$ if $\alpha_i \leq \rho$ for $1 \leq i \leq n$, in our setting this stronger condition is not needed. The data in the network should be such that the accessibility is not discontinued, which means that if a process is allowed to access the final edge of a path it must by allowed to access all ancestor edges. Similarly, a process with roles ρ can access a tree path $\texttt{a}_1^{\tau_1}/\ldots/\texttt{a}_n^{\tau_n}$ if $\tau_n \leq \rho$. Now, we can define:

Definition 3.2.7 (Access to data terms). *A process with roles can access a data term* V *in a data tree* D *if it can access the tree path from the root of* D *to* V.

Example 3.2.8. Let as assume that data $D_{\texttt{mb}}$ at the location `musicbox` from Example 3.2.5 is as follows:

$$D_{\texttt{mb}} = \texttt{music}^{\{\texttt{guest},\top\}}[\texttt{song}^{\{\texttt{guest},\top\}}[D_{\texttt{demo}}] \mid \texttt{song}^{\{\texttt{member},\top\}}[D_{\texttt{full}}] \mid \texttt{help}^{\{\texttt{owner},\top\}}[\Box R]]$$

Data $D_{\texttt{mb}}$ resembles to data of Example 3.2.4. Here, we have left some parts of data unspecified, but we have specified annotations on the edges and there is an additional part of data, $\texttt{help}^{\{\texttt{owner},\top\}}[\Box R]$.

A process with role `member` is allowed to access existing data tree paths

$$\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{guest},\top\}} \quad \text{and} \quad \texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}$$

because

$$\{\texttt{guest}, \top\} \leq \{\texttt{member}\} \quad \text{and} \quad \{\texttt{member}, \top\} \leq \{\texttt{member}\}.$$

This also means that such process can access data terms $D_{\texttt{demo}}$ and $D_{\texttt{full}}$. The same process would be allowed to access the path $\texttt{music}^{\{\texttt{member},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}$, which is not, currently, a path in the data tree. A process with role `guest` is allowed to access the path $\texttt{music}^{\{\texttt{member},\top\}}/\texttt{song}^{\{\texttt{guest},\top\}}$, which is also not a path in the data tree $D_{\texttt{mb}}$. A process with roles $\{\texttt{guest}, \texttt{administrator}\}$ is not allowed to access the path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}$. A process with role `owner` is allowed to access the data tree path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{help}^{\{\texttt{owner},\top\}}$ and data term $\Box R$ at the end of the path. Processes with lower roles are not allowed to access this data term. A process with role `owner` is allowed to access the paths $\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}}$ and $\texttt{music}^{\{\texttt{member}\}}/\texttt{song}^{\{\texttt{guest}\}}$. △

The next lemma shows that an edge annotated with the empty set of roles would be inaccessible for all processes.

Lemma 3.2.9. *A set ρ such that $\emptyset \leq \rho$ does not exist.*

Proof. Obviously, by Definition 3.2.1. □

Recall that, in our calculus, all edges in a data tree are annotated with sets of roles which contain role $\top$ and that a well-formed policy does not allow $\top$ to be disabled.

The next lemma states that a processes with role $\top$ can access all data, since the sets of roles associated to tree edges are never empty.

Lemma 3.2.10. *Let $\top \in \rho$. If $\sigma \neq \emptyset$, then $\sigma \leq \rho$.*

Proof. Obviously, by Definition 3.2.1. □

The paths are used to identify data terms in data trees. We say that a path edge $\mathtt{a}^{\alpha}$ *complies* with a tree edge $\mathtt{a}^{\tau}$ if for each role of α there is at least one smaller or equal role in τ, i.e. if set α complies with set τ. More formally:

Definition 3.2.11 (Edge compliance). *Edge $\mathtt{a}^{\alpha}$ complies with edge $\mathtt{a}^{\tau}$ if $\tau \leqq \alpha$.*

We extend this definition to paths and say that path $\mathtt{a}_1^{\alpha_1}/\ldots/\mathtt{a}_n^{\alpha_n}$ complies with a tree path $\mathtt{a}_1^{\tau_1}/\ldots/\mathtt{a}_n^{\tau_n}$ if $\tau_i \leqq \alpha_i$ for $1 \leq i \leq n$. Path compliance is used to define when a path, as a parameter of a process, identifies a data term in a data tree. This is a stronger notion, which requires stronger conditions, than just checking if there is a permission to access the data term. Namely, we need to ask that every edge of the process path complies with the corresponding edge of the tree path. Now we can define:

Definition 3.2.12 (Identification of data terms). *A path identifies a data term V in a data tree D if it complies with the tree path from the root of D to V.*

The next lemma shows that if a process is allowed to access a path, then that process is allowed to access data identified by that path.

Lemma 3.2.13. *If $\alpha \leq \rho$ and $\tau \leqq \alpha$, then $\tau \leq \rho$.*

Proof. From $\alpha \leq \rho$ and Definition 3.2.1, we know that there exist a role $s \in \alpha$ and a role $r \in \rho$ such that $s \sqsubseteq r$. Then, from $\tau \leqq \alpha$ and the same definition we obtain that there exists a role $t \in \tau$ such that $t \sqsubseteq s$. We get, by transitivity of $\sqsubseteq$, that there exist two roles $t \in \tau$ and $r \in \rho$ such that $t \sqsubseteq r$, which implies $\tau \leq \rho$. □

Example 3.2.14. The path $\texttt{music}^{\{\texttt{member}\}}/\texttt{song}^{\{\texttt{member}\}}$ complies with two paths in $D_{\texttt{mb}}$ of Example 3.2.8. Namely, the paths $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{guest},\top\}}$ and $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}$. So, it identifies two data terms $D_{\texttt{demo}}$ and $D_{\texttt{full}}$. This path is accessible only to processes with roles ρ such that $\{\texttt{member}\} \leq \rho$, i.e. processes which have at least the role `member` or one higher role.

The path $\texttt{music}^{\{\texttt{guest}\}}/\texttt{song}^{\{\texttt{guest}\}}$ identifies just the data term $D_{\texttt{demo}}$ because it complies only with the path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{guest},\top\}}$. This path is accessible to processes with roles ρ such that $\{\texttt{guest}\} \leq \rho$, i.e. processes which have at least the role `guest` or one higher role.

A process with role `guest` is allowed to access a path $\texttt{music}^{\{\texttt{member}\}}/\texttt{song}^{\{\texttt{guest}\}}$. This path complies with the data tree path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{guest},\top\}}$, which identifies just $D_{\texttt{demo}}$.

The process

$$\texttt{READ}_{\texttt{music}^{\{\bot\}}/\texttt{song}^{\{\bot\}}}(\chi).P^{\neg\texttt{member}}$$

is allowed to access the path $\texttt{music}^{\{\bot\}}/\texttt{song}^{\{\bot\}}$, as well as, both data terms $D_{\texttt{demo}}$ and $D_{\texttt{full}}$. On the other hand, path $\texttt{music}^{\{\bot\}}/\texttt{song}^{\{\bot\}}$ does not comply with any path of data tree $D_{\texttt{mb}}$, i.e. this path does not identify data terms in $D_{\texttt{mb}}$. △

Besides paths, processes that aim to read or change data use patterns to identify data. In order to match the pattern, a data term, besides the shape, must satisfy type information provided in the pattern. The type information is such that the data term respects the policy of the location at which it will be used. Namely,

(1) if the pattern is $\Box x^{(\sigma,\mathcal{E},\mathcal{D})}$, then the data term must be a script which can run at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$,

(2) if the pattern is $y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}$, then the data term must be a pointer in which

   (i) the last edge of the path has the set α of roles and

   (ii) the policy of the location is $(\sigma, \mathcal{E}, \mathcal{D})$,

(3) if the pattern is $x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$, then the data term must be a tree

   (i) which can stay at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$ and

   (ii) such that the union of the sets of roles associated to the top edges is τ and

   (iii) such that a process with roles ρ can access the whole tree, if $\zeta \leq \rho$.

These conditions are enforced by using the type assignment system of Section 3.3. More precisely, we define function match:

(1) $\mathsf{match}(\Box x^{(\sigma,\mathcal{E},\mathcal{D})}, \Box R)$ if $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$;

$$D \mid \emptyset_D \equiv D \qquad D_1 \mid D_2 \equiv D_2 \mid D_1 \qquad (D_1 \mid D_2) \mid D_3 \equiv D_1 \mid (D_2 \mid D_3)$$

$$R \mid 0^{\neg\rho} \equiv R \qquad R_1 \mid R_2 \equiv R_2 \mid R_1 \qquad (R_1 \mid R_2) \mid R_3 \equiv R_1 \mid (R_2 \mid R_3)$$

$$(\nu a^{T\mathsf{v}})0^{\neg\rho} \equiv 0^{\neg\rho} \qquad (\nu a^{T\mathsf{v}})(\nu b^{T\mathsf{w}})R \equiv (\nu b^{T\mathsf{w}})(\nu a^{T\mathsf{v}})R$$

$$a^{T\mathsf{v}} \notin \mathsf{fn}(R) \Rightarrow (\nu a^{T\mathsf{v}})(R \mid R') \equiv R \mid (\nu a^{T\mathsf{v}})R' \qquad (*P)^{\neg\rho} \equiv P^{\neg\rho} \mid (*P)^{\neg\rho}$$

$$(P \mid Q)^{\neg\rho} \equiv P^{\neg\rho} \mid Q^{\neg\rho} \qquad N_1 \,[\!]\, N_2 \equiv N_2 \,[\!]\, N_1 \quad (N_1 \,[\!]\, N_2) \,[\!]\, N_3 \equiv N_1 \,[\!]\, (N_2) \,[\!]\, N_3)$$

$$(\nu a^{T\mathsf{v}})(\nu b^{T\mathsf{w}})N \equiv (\nu b^{T\mathsf{w}})(\nu a^{T\mathsf{v}})N$$

$$a^{T\mathsf{v}} \notin \mathsf{fn}(N) \Longrightarrow (\nu a^{T\mathsf{v}})(N \,[\!]\, N') \equiv N \,[\!]\, (\nu a^{T\mathsf{v}})N'$$

$$D \equiv D' \wedge R \equiv R' \Longrightarrow l[D \parallel R] \equiv l[D' \parallel R']$$

$$a^{T\mathsf{v}} \notin \mathsf{fn}(D) \Longleftrightarrow l[D \parallel (\nu a^{T\mathsf{v}})R] \equiv (\nu a^{T\mathsf{v}})l[D \parallel R]$$

Figure 3.9: Structural congruence

(2) $\mathsf{match}(y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}, p@l)$ if $\emptyset \vdash p : \mathit{Path}(\alpha)$ and $\emptyset \vdash l : \mathit{Location}(\sigma,\mathcal{E},\mathcal{D})$;

(3) $\mathsf{match}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, D)$ if $\emptyset \vdash D : \mathit{DataTree}(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)$.

3.2.3 Semantics

The operational semantics is defined in terms of a structural congruence over data, processes and networks, an interaction relation and a reduction relation. Structural congruence identifies structurally equivalent terms. It is the smallest relation $\equiv$ including alpha conversion and the laws in Figure 3.9, stating that parallel composition of data, processes with roles and networks is commutative and associative; that parallel composition of data and processes have $\emptyset_D$ and $0^{\neg\rho}$ as neutral elements, respectively; that a replicated process with roles may be unfolded; and that the same properties hold for the restriction operator as in π-calculus, taking into account that processes with roles may share communication channels at the same or at different locations. Free and bound names of data trees, processes and networks are defined naturally by extending the definition in Figure 2.2 and taking into account that the binders of the calculus are input, restriction and commands for reading and changing data. Variables may be bound by inputs and commands for reading and changing. We assume in process $\mathtt{CHANGE}_p(\chi,\mathsf{V}).P$ pattern variables are bound in V but not in P. Restriction binds channel names.

The operational semantics for data and roles management depends on auxiliary functions given in Figure 3.10. These functions select the exact part of the data that is identified by one path and one pattern, using the relations of Definition 3.2.1, and do the changes if required. Besides others, they all have a data tree and a path as arguments. Essentially, the functions start the computation at

$$
\begin{array}{lcl}
\mathtt{run}(\emptyset_D, p) & = & \emptyset \\
\mathtt{run}(D_1 \mid D_2, p) & = & \mathtt{run}(D_1, p) \cup \mathtt{run}(D_2, p) \\
\mathtt{run}(\mathsf{b}^\tau[\mathsf{V}], p) & = & \begin{cases} \{R\} & \text{if } p = \mathsf{b}^\alpha \text{ and } \tau \leqq \alpha \text{ and } \mathsf{V} = \square R \\ \mathtt{run}(\mathsf{V}, q) & \text{if } p = \mathsf{b}^\alpha/q \text{ and } \tau \leqq \alpha \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{read}(\emptyset_D, p, \chi) & = & \emptyset \\
\mathtt{read}(D_1 \mid D_2, p, \chi) & = & \mathtt{read}(D_1, p, \chi) \cup \mathtt{read}(D_2, p, \chi) \\
\mathtt{read}(\mathsf{b}^\tau[\mathsf{V}], p, \chi) & = & \begin{cases} \{\!\{\mathsf{V}/|\chi|\}\!\} & \text{if } p = \mathsf{b}^\alpha \text{ and } \tau \leqq \alpha \text{ and } \mathsf{match}(\chi, \mathsf{V}) \\ \mathtt{read}(\mathsf{V}, q, \chi) & \text{if } p = \mathsf{b}^\alpha/q \text{ and } \tau \leqq \alpha \\ & \quad \text{and } \mathsf{V} \text{ is a data tree} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{change}(\emptyset_D, p, \chi, \mathsf{W}) & = & \emptyset_D \\
\mathtt{change}(D_1 \mid D_2, p, \chi, \mathsf{W}) & = & \mathtt{change}(D_1, p, \chi, \mathsf{W}) \mid \mathtt{change}(D_2, p, \chi, \mathsf{W}) \\
\mathtt{change}(\mathsf{b}^\tau[\mathsf{V}], p, \chi, \mathsf{W}) & = & \begin{cases} \mathsf{b}^\tau[\mathsf{W}\{\!\{\mathsf{V}/|\chi|\}\!\}] & \text{if } p = \mathsf{b}^\alpha \text{ and } \tau \leqq \alpha \\ & \quad \text{and } \mathsf{match}(\chi, \mathsf{V}) \\ \mathsf{b}^\tau[\mathtt{change}(\mathsf{V}, q, \chi, \mathsf{W})] & \text{if } p = \mathsf{b}^\alpha/q \text{ and } \tau \leqq \alpha \\ & \quad \text{and } \mathsf{V} \text{ is a data tree} \\ \mathsf{b}^\tau[\mathsf{V}] & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{enable}(\emptyset_D, p, r) & = & \emptyset_D \\
\mathtt{enable}(D_1 \mid D_2, p, r) & = & \mathtt{enable}(D_1, p, r) \mid \mathtt{enable}(D_2, p, r) \\
\mathtt{enable}(\mathsf{b}^\tau[\mathsf{V}], p, r) & = & \begin{cases} \mathsf{b}^\tau[\mathsf{V}^{+r}] & \text{if } p = \mathsf{b}^\alpha \text{ and } \tau \leqq \alpha \\ \mathsf{b}^\tau[\mathtt{enable}(\mathsf{V}, q, r)] & \text{if } p = \mathsf{b}^\alpha/q \text{ and } \tau \leqq \alpha \\ & \quad \text{and } \mathsf{V} \text{ is a data tree} \\ \mathsf{b}^\tau[\mathsf{V}] & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{disable}(\emptyset_D, p, r) & = & \emptyset_D \\
\mathtt{disable}(D_1 \mid D_2, p, r) & = & \mathtt{disable}(D_1, p, r) \mid \mathtt{disable}(D_2, p, r) \\
\mathtt{disable}(\mathsf{b}^\tau[\mathsf{V}], p, r) & = & \begin{cases} \mathsf{b}^\tau[\mathsf{V}^{-r}] & \text{if } p = \mathsf{b}^\alpha \text{ and } \tau \leqq \alpha \\ \mathsf{b}^\tau[\mathtt{disable}(\mathsf{V}, q, r)] & \text{if } p = \mathsf{b}^\alpha/q \text{ and } \tau \leqq \alpha \\ & \quad \text{and } \mathsf{V} \text{ is a data tree} \\ \mathsf{b}^\tau[\mathsf{V}] & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
(\emptyset_D)^{+r} = \emptyset_D & (\emptyset_D)^{-r} = \emptyset_D \\
(D_1 \mid D_2)^{+r} = (D_1)^{+r} \mid (D_2)^{+r} & (D_1 \mid D_2)^{-r} = (D_1)^{-r} \mid (D_2)^{-r} \\
(\mathsf{a}^\tau[\mathsf{V}])^{+r} = \mathsf{a}^{\tau \cup \{r\}}[\mathsf{V}] & (\mathsf{a}^\tau[\mathsf{V}])^{-r} = \mathsf{a}^{\tau \setminus r}[(\mathsf{V})^{-r}] \\
(p@l)^{+r} = p@l & (p@l)^{-r} = p@l \\
(\square R)^{+r} = \square R & (\square R)^{-r} = \square R
\end{array}
$$

Figure 3.10: Definitions of auxiliary functions

root of the data tree and try to follow the given path. If such a path exists in the data tree and the given path complies with it, the functions operate on the data terms at the end of the tree path. It is easy to check that selected data terms are indeed identified by the paths, i.e. if a function operates on a data term, the path which is the argument of the function complies with the tree path leading to this data term. The functions `run` and `read` return a set, while the functions `change`, `enable` and `disable` return a data tree. Each of them has three cases in the definition:

- if the data tree is the empty data tree, the result is the empty set or the empty data tree;
- if the data tree is a parallel composition of subtrees, then the function is applied to both subtrees and the result is the union or the parallel composition of the values computed on subtrees;
- otherwise the function checks if the top path edge complies with the top tree edge and:
  - in case of compliance
    - if the path has only one edge, then the data term is identified by the path. Some functions do additional checks and changes;
    - if the path is longer, the function is applied recursively;
  - in case of non-compliance the result is the empty set or the current data tree.

In particular, the function

- `run` checks if the identified data term is a script. The final result is the set which contains all found processes.

- `read` has an additional argument, a pattern χ, and checks if the identified data term matches the pattern. It returns a set of substitutions of the shape $\{\!\{\mathsf{V}/|\chi|\}\!\}$, which are such that V matches the given pattern χ. If the data term and the patterns are pointers, the substitution replaces the location variable with the location and the path variable with the path. If the data term and the pattern are scripts, the substitution replaces the variable with the process. If the data term and the pattern are data trees, the substitution replaces the variable with the data tree. More formally:

  Definition 3.2.15. *For given data term* V *and pattern* χ*, we define* matching substitutions*,* $\{\!\{\mathsf{V}/\chi\}\!\}$*, as follows:*
  $\{\!\{p@l/y@x\}\!\} = \{p/y, l/x\}$
  $\{\!\{\Box R/\Box x\}\!\} = \{R/x\}$
  $\{\!\{D/x\}\!\} = \{D/x\}$

- `change` has two arguments, other than the data tree and the path, a pattern χ and another data term W, and checks if the identified data term matches the pattern. In case of match, the function substitutes the pattern with the identified data term and then put this new term instead of the one identified by the path.

- `enable` and `disable` have an extra argument, a role r, and they use the auxiliary functions $^{+r}$ and $^{-r}$, respectively, to change the sets of roles assigned to the edges of identified data terms. The function $^{+r}$ adds the role r to the initial tree edges starting from the roots of the subtrees identified by the given path, if any, and it does nothing otherwise. The function $^{-r}$ removes all the roles less than or equal to the role r from all the edges in the subtrees identified by the given path, if any, and it does nothing otherwise.

In the following examples refer to data tree $D_{\mathtt{mb}}$ of Example 3.2.8.

Example 3.2.16. Let us assume that there is a pattern $x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$ such that

$$\mathsf{match}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, D_{\mathtt{demo}}) \quad \text{and} \quad \mathsf{match}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, D_{\mathtt{full}}).$$

By the definition of functions, given in Figure 3.10, we can derive:

$$\begin{aligned}&\mathtt{read}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{member}\}}/\mathtt{song}^{\{\mathtt{member}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)})\\ &= \mathtt{read}(D_{\mathtt{mb}}, \mathtt{song}^{\{\mathtt{member}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)})\\ &= \{\{\!\{D_{\mathtt{demo}}/x\}\!\}, \{\!\{D_{\mathtt{full}}/x\}\!\}\}.\end{aligned}$$

Similarly,

$$\begin{aligned}&\mathtt{read}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}/\mathtt{song}^{\{\mathtt{guest}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)})\\ &= \mathtt{read}(D_{\mathtt{mb}}, \mathtt{song}^{\{\mathtt{guest}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)})\\ &= \{\{\!\{D_{\mathtt{demo}}/x\}\!\}\}.\end{aligned}$$

The auxiliary functions defined in Figure 3.10 have as arguments a data tree and a path. Some of them have a pattern as argument. In case the path on input does not comply with any path in the data tree or in case the identified data does not match the pattern, functions `run` and `read` return $\emptyset$ and the rest of auxiliary function defined in Figure 3.10 return the same data tree as the one on the input. We give examples for the function `read` :

$$\begin{aligned}&\mathtt{read}(D_{\mathtt{mb}}, \mathtt{song}^{\{\mathtt{guest}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}) = \emptyset\\ &\mathtt{read}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}/\mathtt{help}^{\{\mathtt{owner}\}}/\mathtt{help}^{\{\mathtt{owner}\}}, \Box x^{(\sigma,\mathcal{E},\mathcal{D})}) = \emptyset\\ &\mathtt{read}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}/\mathtt{help}^{\{\mathtt{owner}\}}, y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}) = \emptyset.\end{aligned}$$

△

Example 3.2.17. Let us now specify data terms $D_{\mathtt{demo}}$ and $D_{\mathtt{full}}$ of Example 3.2.8:

$$D_{\mathtt{demo}} = \mathtt{title}^{\{\mathtt{guest},\top\}}[\mathtt{Yellow\ sun}] \mid \mathtt{download}^{\{\mathtt{guest},\top\}}[\mathtt{ys}^{\{\mathtt{guest}\}}@\mathtt{repository}]$$

$$D_{\mathtt{full}} = \mathtt{title}^{\{\mathtt{guest},\top\}}[\mathtt{Blue\ moon}] \mid \mathtt{download}^{\{\mathtt{member},\top\}}[\mathtt{bm}^{\{\mathtt{member}\}}@\mathtt{repository}]$$

If we assume

$$\mathsf{match}(y^{\{\mathtt{guest}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')}, \mathtt{ys}^{\{\mathtt{guest}\}}@\mathtt{repository}),$$

we can derive:

$$\begin{aligned}&\mathtt{read}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}/\mathtt{song}^{\{\mathtt{guest}\}}/\mathtt{download}^{\{\mathtt{guest}\}}, y^{\{\mathtt{guest}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')})\\ &= \mathtt{read}(D_{\mathtt{mb}}, \mathtt{song}^{\{\mathtt{guest}\}}/\mathtt{download}^{\{\mathtt{guest}\}}, y^{\{\mathtt{guest}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')})\\ &= \mathtt{read}(D_{\mathtt{mb}}, \mathtt{download}^{\{\mathtt{guest}\}}, y^{\{\mathtt{guest}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')})\\ &= \{\!\{\mathtt{download}^{\{\mathtt{guest}\}}[\mathtt{ys}^{\{\mathtt{guest}\}}@\mathtt{repository}]/y@x\}\!\}\end{aligned}$$

△

We give further examples of results computed by auxiliary functions:

Example 3.2.18. The data tree $D_{\mathtt{mb}}$ has the scripted process R at the end of path

$$\mathtt{music}^{\{\mathtt{guest},\top\}}/\mathtt{help}^{\{\mathtt{owner},\top\}}.$$

The path $\mathtt{music}^{\{\mathtt{owner}\}}/\mathtt{help}^{\{\mathtt{owner}\}}$ complies with the path $\mathtt{music}^{\{\mathtt{guest},\top\}}/\mathtt{help}^{\{\mathtt{owner},\top\}}$. As expected,

$$\mathtt{run}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{owner}\}}/\mathtt{help}^{\{\mathtt{owner}\}}) = \{R\}.$$

If the path does not comply with any path in the data tree the function run returns $\emptyset$. For example:

$$\mathtt{run}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{member}\}}/\mathtt{help}^{\{\mathtt{member}\}}) = \emptyset$$
$$\mathtt{run}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{owner}\}}/\mathtt{help}^{\{\mathtt{owner}\}}/\mathtt{help}^{\{\mathtt{owner}\}}) = \emptyset.$$

The function run returns the same result if the data at the end of the path is not a script. For example:

$$\mathtt{run}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{owner}\}}) = \emptyset$$
$$\mathtt{run}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{owner}\}}/\mathtt{song}^{\{\mathtt{owner}\}}/\mathtt{download}^{\{\mathtt{owner}\}}) = \emptyset.$$

△

Example 3.2.19. If we assume that the data at the end of tree path which complies with the path $\mathtt{music}^{\{\mathtt{guest}\}}$ matches pattern $x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$, then

$$\mathtt{change}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}, x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, \emptyset_D) = \mathtt{music}^{\{\mathtt{guest},\top\}}[\emptyset_D].$$

If we assume

$$\mathsf{match}(y^{\{\mathtt{member}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')}, \mathtt{bm}^{\{\mathtt{member}\}}@\mathtt{repository}),$$

then

$$\mathtt{change}(D_{\mathtt{mb}}, \mathtt{music}^{\{\mathtt{guest}\}}/\mathtt{song}^{\{\mathtt{member}\}}/\mathtt{download}^{\{\mathtt{member}\}}, y^{\{\mathtt{member}\}}@x^{(\sigma',\mathcal{E}',\mathcal{D}')}, \mathsf{V})$$
$$= \mathtt{music}^{\{\mathtt{guest},\top\}}[\mathtt{song}^{\{\mathtt{guest},\top\}}[D_{\mathtt{demo}}] \mid \mathtt{song}^{\{\mathtt{member},\top\}}[D'_{\mathtt{full}}] \mid \mathtt{help}^{\{\mathtt{owner},\top\}}[\Box R]].$$

The change affects only the pointer at the end of the data tree path

$$\mathtt{music}^{\{\mathtt{guest},\top\}}/\mathtt{song}^{\{\mathtt{member},\top\}}/\mathtt{download}^{\{\mathtt{member},\top\}}.$$

In particular, it replaces the identified pointer with the new data term V on which the substitution $\{\!\{\mathtt{bm}^{\{\mathtt{member}\}}@\mathtt{repository}/y@x\}\!\}$ is applied. So,

$$D'_{\mathtt{full}} = \mathtt{title}^{\{\mathtt{guest},\top\}}[\mathtt{Blue\ moon}] \mid \mathtt{download}^{\{\mathtt{guest},\top\}}[\mathsf{W}]$$

where

$$\mathsf{W} = \mathsf{V}\{\!\{\mathtt{bm}^{\{\mathtt{member}\}}@\mathtt{repository}/y@x\}\!\}.$$

△

Example 3.2.20. By means of function enable we can extend the access rights of processes with role guest:

$$\begin{array}{l}\texttt{enable}(D_{\texttt{mb}}, \texttt{music}^{\{\texttt{guest}\}}/\texttt{song}^{\{\texttt{member}\}}, \texttt{guest})\\ = \texttt{music}^{\{\texttt{guest},\top\}}[\texttt{song}^{\{\texttt{guest},\top\}}[D''_{\texttt{demo}}] \mid \texttt{song}^{\{\texttt{member},\top\}}[D''_{\texttt{full}}] \mid \texttt{help}^{\{\texttt{owner},\top\}}[\Box R]]\end{array}$$

Data tree $D''_{\texttt{demo}}$ is the same as $D_{\texttt{demo}}$ because role guest was already allowed to access $D_{\texttt{demo}}$. In fact, the only edge that should have changed annotations is $\texttt{download}^{\{\texttt{member},\top\}}$, so:

$$D''_{\texttt{full}} = \texttt{title}^{\{\texttt{guest},\top\}}[\texttt{Blue moon}]|\texttt{download}^{\{\texttt{guest},\texttt{member},\top\}}[\texttt{bm}^{\{\texttt{member}\}}@\texttt{repository}]$$

The function $^{+r}$ adds role r just to the initial edges of its argument, so:

$$\begin{array}{l}\texttt{enable}(D_{\texttt{mb}}, \texttt{music}^{\{\texttt{guest}\}}, \texttt{guest})\\ =\\ \texttt{music}^{\{\texttt{guest},\top\}}[\texttt{song}^{\{\texttt{guest},\top\}}[D_{\texttt{demo}}] \mid \texttt{song}^{\{\texttt{guest},\texttt{member},\top\}}[D_{\texttt{full}}] \mid \texttt{help}^{\{\texttt{guest},\top\}}[\Box R]]\end{array}$$

The function $^{-r}$ removes role r from all the edges of its argument, so:

$$\begin{array}{l}\texttt{disable}(D_{\texttt{mb}}, \texttt{music}^{\{\texttt{guest}\}}, \texttt{member})\\ = \texttt{music}^{\{\texttt{guest},\top\}}[\texttt{song}^{\{\top\}}[D'''_{\texttt{demo}}] \mid \texttt{song}^{\{\top\}}[D'''_{\texttt{full}}] \mid \texttt{help}^{\{\texttt{owner},\top\}}[\Box R]]\end{array}$$

where

$$D'''_{\texttt{demo}} = \texttt{title}^{\{\top\}}[\texttt{Yellow sun}] \mid \texttt{download}^{\{\top\}}[\texttt{ys}^{\{\texttt{guest}\}}@\texttt{repository}]$$

and

$$D'''_{\texttt{full}} = \texttt{title}^{\{\top\}}[\texttt{Blue moon}] \mid \texttt{download}^{\{\top\}}[\texttt{bm}^{\{\texttt{member}\}}@\texttt{repository}].$$

△

We define an interaction relation $\rightsquigarrow$ in order to describe local communication of processes with roles and their interaction with the local data tree. It is given by the rules in Figure 3.11. Rules [L-Structural], [L-Parallel], [L-Restriction] and [L-Communication] are essentially the same as [R-Structural], [R-Parallel], [R-Restriction] and [R-Communication] reduction rules for π-calculus, given in Figure 2.5, except that they refer to both processes and data. Processes with different roles may communicate. Parallel composition of processes may change the data.

In rule [L-Run], the process $\texttt{RUN}_p{}^{\neg\rho}$ activates all the scripts identified by the path p in the data tree D, using the function run. The parallel composition of the activated processes is then executed.

In rule [L-Read], the process $\texttt{READ}_p(\chi).P^{\neg\rho}$ obtains a set of substitutions using the function read, and for each substitution s in this set activates process $P\mathsf{s}^{\neg\rho}$. These substitutions are are identified by the path p and the pattern χ in the local data tree D.

In rule [L-Change], the process $\texttt{CHANGE}_p(\chi, \mathsf{V}).P^{\neg\rho}$ modifies the data tree D using the function change. Each data term W identified by the path p and matching the pattern χ gets replaced by $\mathsf{V}\{\mathsf{W}/|\chi|\}$.

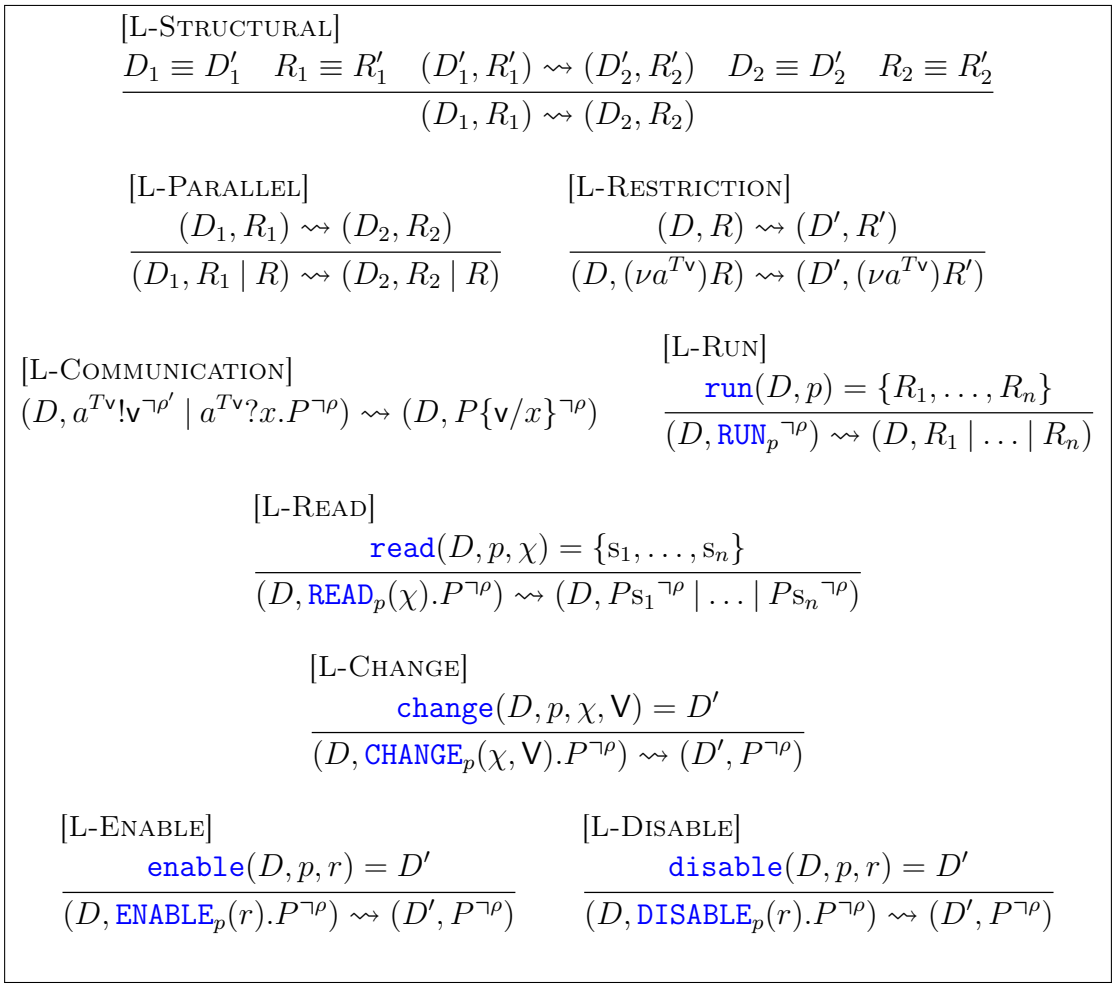

Figure 3.11: Local communication and interaction rules

$$\text{[R-Structural]}\quad \frac{N_1 \equiv N_1' \quad N_1' \to N_2' \quad N_2' \equiv N_2}{N_1 \to N_2} \qquad \text{[R-Parallel]}\quad \frac{N_1 \to N_2}{N_1 \,[]\, N \to N_2 \,[]\, N}$$

$$\text{[R-Restriction]}\quad \frac{N \to N'}{(\nu a^{T_\mathsf{v}})N \to (\nu a^{T_\mathsf{v}})N'} \qquad \text{[R-Local]}\quad \frac{(D, R) \rightsquigarrow (D', R')}{l[D \parallel R] \to l[D' \parallel R']}$$

$$\text{[R-Go]}\quad l[D_l \parallel (\texttt{GO}\ m.R)^{\neg\rho} \mid R_l] \,[]\, m[D_m \parallel R_m] \to l[D_l \parallel R_l] \,[]\, m[D_m \parallel R \mid R_m]$$

$$\text{[R-Stay]}\quad l[D \parallel (\texttt{GO}\ l.R)^{\neg\rho} \mid R'] \to l[D \parallel R \mid R']$$

Figure 3.12: Reduction rules

In rule [L-Enable], the process $\texttt{ENABLE}_p(r).P^{\neg\rho}$ adds the role r to the edges starting from the roots of subtrees identified by the path p, using the function `enable`.

In rule [L-Disable], the process $\texttt{DISABLE}_p(r).P^{\neg\rho}$ removes the role r from all the edges in the subtrees identified by the path p, using the function `disable`.

Reduction relation is the smallest relation between networks defined by the rules in Figure 3.12. Rules [R-Structural], [R-Parallel] and [R-Restriction] are standard rules which formalize the property that the reduction relation is defined up to structural congruence and closed with respect to the usual static contexts, as in the π-calculus.

The rest of reduction rules reflect the fact that processes may be involved in local interactions or they may reallocate. In particular, rule [R-Local] describes local communication and interaction using the interaction relation.

Rule [R-Go] describes the migration of a process from one location to another one, while rule [R-Stay] describes a migration to the current location.

We use $\to^*$ to denote the reflexive and transitive closure of $\to$.

We now give examples of interactions and reductions in the network

$$\texttt{musicbox}[D_{\texttt{mb}} \parallel R_{\texttt{mb}}] \,[]\, \texttt{repository}[D_{\texttt{rp}} \parallel R_{\texttt{rp}}]$$

specified in Examples 3.2.5, 3.2.8 and 3.2.17. For simplicity, we assume that variables x and y are not free in P, so the substitutions of these variables do not affect P.

Example 3.2.21. Using the result of function `read` given in Example 3.2.17, by [L-Read], we have:

$$(D_{\texttt{mb}}, \texttt{READ}_{\texttt{music}^{\{\texttt{guest}\}}/\texttt{song}^{\{\texttt{guest}\}}/\texttt{download}^{\{\texttt{guest}\}}}(y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}).\texttt{GO}\ x.P^{\neg\texttt{guest}})^{\neg\texttt{member}})$$
$$\rightsquigarrow (D_{\texttt{mb}}, (\texttt{GO}\ \texttt{repository}.P^{\neg\texttt{guest}})^{\neg\texttt{member}}).$$

If we denote

$$R_{\texttt{mb}} = \texttt{READ}_{\texttt{music}^{\{\texttt{guest}\}}/\texttt{song}^{\{\texttt{guest}\}}/\texttt{download}^{\{\texttt{guest}\}}}(y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}).\texttt{GO}\ x.P^{\neg\texttt{guest}})^{\neg\texttt{member}},$$

we can, by rules [R-Local], [R-Parallel], [R-Structural] and [R-Go], derive

$$\begin{aligned}&\texttt{musicbox}[D_{\texttt{mb}} \parallel R_{\texttt{mb}}] \,[]\, \texttt{repository}[D_{\texttt{rp}} \parallel R_{\texttt{rp}}]\\ \to^* &\texttt{musicbox}[D_{\texttt{mb}} \parallel (\texttt{GO}\ \texttt{repository}.P^{\neg\texttt{guest}})^{\neg\texttt{member}}] \,[]\, \texttt{repository}[D_{\texttt{rp}} \parallel R_{\texttt{rp}}]\\ \to^* &\texttt{musicbox}[D_{\texttt{mb}} \parallel 0^{\neg\rho}] \,[]\, \texttt{repository}[D_{\texttt{rp}} \parallel P^{\neg\texttt{guest}} \mid R_{\texttt{rp}}]\end{aligned}$$

△

Example 3.2.22. By [L-Communication], we have

$$\begin{aligned}&(D_{\texttt{mb}}, a^{T\texttt{v}}!\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}\neg\rho'} \mid a^{T\texttt{v}}?x.\texttt{READ}_x(\chi).P^{\neg\texttt{guest}})\\ &\rightsquigarrow (D_{\texttt{mb}}, \texttt{READ}_{\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}}}(\chi).P^{\neg\texttt{guest}}).\end{aligned}$$

If we denote

$$R_{\texttt{mb}} = a^{T\texttt{v}}!\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}\neg\rho'} \mid a^{T\texttt{v}}?x.\texttt{READ}_x(\chi).P^{\neg\texttt{guest}},$$

by [R-Local] we can derive

$$\texttt{musicbox}[D_{\texttt{mb}} \parallel R_{\texttt{mb}}] \to \texttt{musicbox}[D_{\texttt{mb}} \parallel \texttt{READ}_{\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}}}(\chi).P^{\neg\texttt{guest}}]$$

△

Example 3.2.23. Using the first result of function `change` given in Example 3.2.19, by [L-Change], we have:

$$(D_{\texttt{mb}}, \texttt{CHANGE}_{\texttt{music}^{\{\texttt{guest}\}}}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, \emptyset_D)^{\neg\texttt{guest}}) \rightsquigarrow (\texttt{music}^{\{\texttt{guest},\top\}}[\emptyset_D], 0^{\neg\texttt{guest}}).$$

If we denote

$$R_{\texttt{mb}} = \texttt{CHANGE}_{\texttt{music}^{\{\texttt{guest}\}}}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, \emptyset_D)^{\neg\texttt{guest}},$$

we can, by rule [R-Local], derive

$$\texttt{musicbox}[D_{\texttt{mb}} \parallel R_{\texttt{mb}}] \to \texttt{musicbox}[\texttt{music}^{\{\texttt{guest},\top\}}[\emptyset_D] \parallel 0^{\neg\texttt{guest}}].$$

△

Example 3.2.24. Let $R = \texttt{ENABLE}_{\texttt{music}^{\{\texttt{guest}\}}}(\texttt{guest})^{\neg\texttt{guest}}$. Using the first result of function `run` given in Example 3.2.18, by [R-Local], we can derive:

$$\begin{aligned}&\texttt{musicbox}[D_{\texttt{mb}} \parallel \texttt{RUN}_{\texttt{music}^{\{\texttt{owner}\}}/\texttt{help}^{\{\texttt{owner}\}}}]\\ &\to \texttt{musicbox}[D_{\texttt{mb}} \parallel \texttt{ENABLE}_{\texttt{music}^{\{\texttt{guest}\}}}(\texttt{guest})^{\neg\texttt{guest}}].\end{aligned}$$

This location may further reduce using the second result of function `enable` given in Example 3.2.20. △

3.2.4 Well-behaved networks

Before we give the definition of what we consider a well-behaved network, we informally introduce the notion of *characteristic roles* for values in our language. Characteristic roles of a channel are characteristic roles of the value this channel transmits. The set of characteristic roles of a data tree is the union of all sets of roles associated to the initial edges of the data tree. The set of characteristic

roles for the empty data tree is $\{\top\}$. The set of characteristic roles of a path is the set of roles associated to the final edge of the path. The set of characteristic roles of a script and a location is the set containing just $\bot$. We denote the set of characteristic roles of a value v with $\mathcal{C}(\mathsf{v})$.

We motivate these choices of characteristic roles by the ability of processes to access data. If a process is allowed to access any part of a data tree, then it must be allowed to access one of the tree's initial edges and if a process is allowed to access the final edge of a path, then it is allowed to access the data at the end of this path. This motivates the choices of characteristic roles for paths and non-empty data tree. As the accessibility of data should not be discontinued, empty data trees, which may be found in the leaves, must be protected by the top role. Implicitly, such trees may be replaced only by a process with the top role. We aim to protect scripts and pointers by their parent edge. The process allowed to access them should already be allowed to access the parent edge, so we say that the set of characteristic roles of scripts and locations is the least protective set, i.e. $\{\bot\}$. In Section 3.3 we relate all values and their types and formalize this notion for value types.

Informally, in a well-behaved network at any location:

(1) a data tree cannot contain roles that are lower than those prescribed as minimal by the location policy and a process must be allowed to access the same minimal roles;

(2) a process with roles can add a role to an edge in the local tree only if this is allowed by the location policy;

(3) a process with roles can erase a role from a subtree of the local tree only if this is allowed by the location policy;

(4) data accessibility should not be discontinued, i.e. if a process is allowed to access an edge in a tree path, then it must be allowed to access its parent;

(5) an edge in a data tree is never completely unavailable, i.e. it is always annotated with a non-empty set of roles;

(6) a process with roles can communicate only values with characteristic roles accessible to the roles of the process;

(7) a process with roles looks for a path in the local tree only if the path is accessible to the process;

(8) a process with roles can get a data term in the local tree only if the data is accessible to the process;

(9) a process with roles can erase a subtree of data only if it can access the whole subtree.

The following definition formalizes what we consider as a secure network.

Definition 3.2.25. *Let* $N \to^* \vec{\nu}(l[D \parallel P^{\neg\rho} \mid R] \,[]\, N')$ *and* $\mathcal{P}(l) = (\sigma, \mathcal{E}, \mathcal{D})$. *We say that* N *is well behaved if the following conditions hold:*

(1) $\sigma \leqq \mathcal{C}(D)$ *and* $\sigma \leq \rho$;

(2) $P = \mathtt{ENABLE}_p(r).Q$ *implies* $(\rho, r) \in^+ \mathcal{E}$;

(3) $P = \mathtt{DISABLE}_p(r).Q$ *implies* $(\rho, r) \in^- \mathcal{D}$.

(4) if $\mathsf{a}_1^{\tau_1}/\ldots/\mathsf{a}_n^{\tau_n}$ *is a path in data tree* D*, then* $\tau_1 \leqq \ldots \leqq \tau_n$ *for every* $i \in \{1, \ldots, n\}$;

(5) if a^{τ} *is an edge in data tree* D*, then* $\tau \neq \emptyset$;

(6) $P = a^{T\mathsf{v}}!\mathsf{v}$ *implies* $\mathcal{C}(\mathsf{v}) \leq \rho$;

(7) $P = \mathtt{RUN}_p$ *or* $P = \mathtt{READ}_p(\chi).Q$ *or* $P = \mathtt{CHANGE}_p(\chi, \mathsf{V}).Q$ *or* $\mathtt{ENABLE}_p(r).Q$ *or* $\mathtt{DISABLE}_p(r).Q$ *implies* $\mathcal{C}(p) \leq \rho$*;*

(8) $P = \mathtt{RUN}_p$ *or* $P = \mathtt{READ}_p(\chi).Q$ *or* $P = \mathtt{CHANGE}_p(\chi, \mathsf{V}).Q$ *or* $\mathtt{ENABLE}_p(r).Q$ *or* $\mathtt{DISABLE}_p(r).Q$ *and* p *identifies a data term* W *in the data tree* D *imply that* W *is accessible to the process* $P^{\neg\rho}$;

(9) $P = \mathtt{CHANGE}_p(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}, \mathsf{V}).Q$ *implies* $\zeta \leq \rho$;

Items (1), (2) and (3) ensure that data and processes with roles respect the security protection policy of their current location. Items (4) and (5) state when we consider data consistent. Items (6), (7), (8) and (9) state what conditions roles of processes should satisfy in order for the processes to send, access and change the data. There is no reason to have additional requirements for the process $\mathtt{GO}\ \lambda.R$ because processes may freely change locations. The the fact that process R respects the policy of location λ will be ensured by the type system presented in the next section.

We now give few examples of ill-behaved locations and networks to illustrate what kind of errors we want to eliminate with the type system presented in the following section. We use notation and results from previous examples and assume

$$\mathcal{P}(\mathtt{repository}) = (\{\mathtt{member}, \mathtt{administrator}\}, (\{\mathtt{owner}\}, \mathtt{administrator}), \emptyset).$$

- Data $D = \mathtt{music}^{\{\mathtt{member}, \top\}}[D']$ and processes $P_1 = a^{T\mathsf{v}}!D_{\mathtt{mb}}{}^{\neg\mathtt{administrator}}$ and $P_2 = \mathtt{RUN}_p{}^{\neg\bot}$ violate item (1) of Definition 3.2.25, for location policy given in Example 3.2.5.

- Process $P^{\neg\mathtt{guest}}$ violates the security policy of location `repository` because $\{\mathtt{member}\} \not\leq \{\mathtt{guest}\}$. For the same reason, process

  $$(\mathtt{GO}\ \mathtt{repository}.P^{\neg\mathtt{guest}})^{\neg\mathtt{member}}$$

  should be eliminated by the type system. Process

  $$(\mathtt{READ}_p(y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}).\mathtt{GO}\ x.P^{\neg\mathtt{guest}})^{\neg\mathtt{member}}$$

  is potentially dangerous, because, for specific p, α and $(\sigma, \mathcal{E}, \mathcal{D})$, as shown in Example 3.2.21, it may obtain a substitution

  $$\{\!\{\mathtt{ys}^{\{\mathtt{guest}\}}@\mathtt{repository}/y@x\}\!\}$$

  and reduce to the previous process.

- Path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}/\texttt{title}^{\{\texttt{guest},\top\}}$ is a path in the data tree $D_{\texttt{mb}}$. It violates item (4) of Definition 3.2.25.

- Process
$$a^{T_{\mathsf{V}}}?x.\texttt{READ}_x(\chi).P^{\neg\texttt{guest}}$$
receives, on channel a, a path. As shown in Example 3.2.22, it may receive
$$\texttt{music}^{\{\texttt{guest}\}}/\texttt{help}^{\{\texttt{owner}\}}$$
which is not a path accessible to the process with role `guest`, and therefore violate item (7) of Definition 3.2.25.

- Process
$$\texttt{CHANGE}_{\texttt{music}^{\{\texttt{guest}\}}}(x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\{\texttt{member}\})},\emptyset_D)^{\neg\texttt{guest}},$$
as shown in Example 3.2.23, erases a subtree which is not fully accessible to it. It violates item (9) of Definition 3.2.25.

- Process
$$\texttt{CHANGE}_{\texttt{music}^{\{\texttt{guest}\}}/\texttt{song}^{\{\texttt{member}\}}/\texttt{download}^{\{\texttt{member}\}}}(y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})},\texttt{a}^{\{\texttt{guest},\top\}}[y@x])^{\neg\texttt{owner}}$$
identifies a pointer and it puts the additional edge $\texttt{a}^{\{\texttt{guest},\top\}}$ before that pointer. In case this action is applied to data $D_{\texttt{mb}}$, as shown in Example 3.2.19, it would create the path
$$\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{song}^{\{\texttt{member},\top\}}/\texttt{download}^{\{\texttt{member},\top\}}/\texttt{a}^{\{\texttt{guest},\top\}}$$
in $D_{\texttt{mb}}$, which would violate item (4) of Definition 3.2.25.

- Process
$$\texttt{RUN}_{\texttt{music}^{\{\texttt{owner}\}}/\texttt{help}^{\{\texttt{owner}\}}},$$
as shown in Example 3.2.24, activates scripted process R at the end of the path $\texttt{music}^{\{\texttt{guest},\top\}}/\texttt{help}^{\{\texttt{owner},\top\}}$ in data tree $D_{\texttt{mb}}$. Process R should be allowed to run at the location `musicbox`. We give three processes that should not be present in the data tree $D_{\texttt{mb}}$ because, in case they are activated, the security of the network will be violated. According to the policy of `musicbox` process
$$\texttt{ENABLE}_p(\texttt{guest})^{\neg\texttt{guest}}$$
does not have roles high enough to enable any other role. It violates item (2) of Definition 3.2.25. Similarly, both processes
$$\texttt{GO repository}.\texttt{ENABLE}_p(\texttt{administrator})^{\neg\texttt{owner}}$$
and
$$\texttt{GO repository}.\texttt{DISABLE}_p(\texttt{administrator})^{\neg\texttt{owner}}$$
would violate the administration policy of location `repository`.

Before we introduce the type system in the following section, we clarify some choices in the language design for which we claim that they do not affect the studied security properties. As already mentioned, we use edge-labelled rooted tree as data model, where edge labels represent tags of XML elements. In the standard XQuery data model [10] tags are nodes. XML documents necessarily have a root element while data trees in our language may have one or more initial edges joined at the root node of the tree. Our paths identify data at the end, not at the last edge. We could be able to identify the whole data tree at a location by introducing a special path with no edges - the empty path. The data is accessible to a process if the tree path is accessible to the process. This means that, if we return the Example 3.2.5, all the data at the end of path $\texttt{music}^{\{\texttt{guest}\}}$ is accessible to a process with role `guest`, regardless of the fact that this role is not allowed to access the end of every path in the identified subtree, i.e. role `guest` is not present at every edge annotation of the identified data. Essentially, if a process is allowed to access an initial edge of a data tree it is allowed to access data, i.e. to read the data. In order to modify the data in any way, it should satisfy additional conditions. We restrict the modifications of data by process $\texttt{CHANGE}_p(\chi, \mathsf{V}).P^{\neg\rho}$ just to processes that are allowed to fully access data (access every edge). So, a process with role `guest` is allowed to access the data identified by the path $\texttt{music}^{\{\texttt{guest}\}}$, but not to change it. One of our goals was to protect data in the leaves - pointers and scripts. In order for a process to access them it must be allowed to access the full path, or more precisely the parent edge. This light condition for changing data could be strengthened by asking, for example that the process is allowed to assess all of the data at the location where the subtree is, meaning that such process could be considered the owner of the location, similarly to what will we discuss in Chapter 4. Another possibility would be to assign richer security protection policies to the locations, which would prescribe exact roles allowed to change data. Conditions for enabling and disabling roles on data tree edges are determined by the policy. The function `enable` adds one role just to the initial edges of the identified subtree, which is more refined than enable the role on the whole subtree.

3.3 Type system

Aiming to prevent security violations, we introduce a type system that enforces well-behavedness of networks.

3.3.1 Types

Syntax of the types given in Figure 3.13. The type $\mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$ denotes the type of a location with policy $(\sigma, \mathcal{E}, \mathcal{D})$. The type $\mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$ denotes the type of a script which can be activated at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$. The type $\mathit{Path}(\alpha)$ denotes the type of a path having the final edge with the set of roles α. The type $\mathit{Pointer}(\alpha)$ denotes the type of a pointer whose path is typed with $\mathit{Path}(\alpha)$. The type $\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ denotes the type of a data tree which can stay at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$, with initial edges asking τ and which can

$\mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$	location type
$\mathit{Pointer}(\alpha)$	pointer type
$\mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$	script type
$\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$	data tree type
$\mathit{Path}(\alpha)$	path type
$\mathit{Channel}(T\mathsf{v})$	channel type
$\mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$	pure process type
$\mathit{Network}$	network type
$\mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$	process with roles type

Figure 3.13: Syntax of the types.

be completely accessed by processes with access to ζ. The type $\mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ denotes the type of a pure process which can stay at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$ and which can be assigned roles ρ. The type $\mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ denotes the type of a process with roles which can stay at locations with policy $(\sigma, \mathcal{E}, \mathcal{D})$. The type $\mathit{Channel}(T\mathsf{v})$ denotes the type of a channel which can communicate values of type $T\mathsf{v}$. We use $T\mathsf{v}$ and $T\mathsf{w}$ to range ranges over *value types*, which are the types that correspond to the values defined in Figure 3.7. The type *Network* denotes the type of a network. We use T to range over types.

A data tree type $\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ is well formed if $\sigma \leqq \tau$, meaning that each role appearing at the initial edges of the data tree has to be bigger than or equal to one role from the set σ of minimal roles which is given by the location policy. This condition implies that each edge in a well-typed data tree has a set of roles which respect the location policy.

A process type $\mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ is well formed if $\sigma \leq \rho$. This requirement guaranties that the process has at least one role bigger than or equal to one role belonging to the set of minimal roles prescribed by the location policy.

In the following we consider only well-formed types.

An environment Γ associates variables with value types and with types of processes with roles, i.e. we define:

$$\Gamma ::= \emptyset \mid \Gamma, x : T\mathsf{v} \mid \Gamma, x : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$$

We use the environment by the standard axioms:

[T- Value Variable] $\quad \Gamma, x : T\mathsf{v} \vdash x : T\mathsf{v}$

[T-Process Variable] $\quad \Gamma, x : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D}) \vdash x : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$

Essentially, a type environment is a finite set of assignments of types to variables. We assume that variables in assignments of type environments are all different. The set of variables of assignments in Γ is called the *domain* of Γ, written $\mathsf{dom}(\Gamma)$. We write $\emptyset$ for the environment whose domain is empty and we write Γ, Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$.

$$\frac{\mathcal{P}(l) = (\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})}\ [\text{T-Location Name}] \qquad \frac{\Gamma \vdash \Pi : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash \Box\Pi : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})}\ [\text{T-Script}]$$

$$\Gamma \vdash \mathsf{a}^{\alpha} : \mathit{Path}(\alpha)\ [\text{T-Path Edge}] \qquad \frac{\Gamma \vdash p : \mathit{Path}(\beta) \quad \Gamma \vdash q : \mathit{Path}(\alpha)}{\Gamma \vdash p/q : \mathit{Path}(\alpha)}\ [\text{T-Path Composition}]$$

$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \Gamma \vdash \lambda : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash p@\lambda : \mathit{Pointer}(\alpha)}\ [\text{T-Pointer}]$$

Figure 3.14: Typing rules for locations, scripts, paths and pointers.

The following nine kinds of *typing judgements* are induced by the syntax of types:

$$\Gamma \vdash \lambda : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D}) \qquad \Gamma \vdash \Box\Pi : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}) \qquad \Gamma \vdash p : \mathit{Path}(\alpha)$$
$$\Gamma \vdash p@\lambda : \mathit{Pointer}(\alpha) \qquad \Gamma \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$$
$$\Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \qquad \Gamma \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$$
$$\Gamma \vdash u : \mathit{Channel}(T\mathsf{v}) \qquad \Gamma \vdash N : \mathit{Network}$$

They state that a term of the calculus is *well typed* in Γ and for sets of roles that appear in the type.

3.3.2 Typing locations, scripts, paths, pointers and data trees

Typing rules for locations, scripts, paths and pointers are given in Figure 3.14. Rule [T-Location Name] states that a location name is well typed in any environment and for the policy assigned to the location name by the function $\mathcal{P}$.

Rule [T-Script] states that a script is well typed in an environment and for a policy if the scripted process is well typed in the same environment and can run at locations with that policy.

Rule [T-Path Edge] states that a path edge with roles α is well typed in any environment and for the set α.

Rule [T-Path Composition] states that a path is well typed in any environment and for the set of roles assigned to its final edge. Notice that this set is, in fact, the set of characteristic roles for the path.

Rule [T-Pointer] states that a pointer $p@\lambda$ is well typed in any environment and for the set of roles α if path p and λ are well typed in the same environment and if α is the set assigned to the final edge of the path.

Typing rules for data trees are given in Figure 3.15. Rule [T-Empty Data] states that an empty data tree is well typed in any environment and for an arbitrary location policy since relation $\sigma \leqq \{\top\}$ holds for any σ.

[T-Empty Data]
$$\Gamma \vdash \emptyset_D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \{\top\}, \{\bot, \top\})$$

[T-Leaf Script]
$$\frac{\Gamma \vdash \Box\Pi : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash \mathtt{a}^{\tau}[\Box\Pi] : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \tau)}$$

[T-Leaf Pointer]
$$\frac{\Gamma \vdash p@\lambda : \mathit{Pointer}(\alpha)}{\Gamma \vdash \mathtt{a}^{\tau}[p@\lambda] : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \tau)}$$

[T-Data Tree]
$$\frac{\Gamma \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta) \quad \tau \leqq \tau'}{\Gamma \vdash \mathtt{a}^{\tau}[D] : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \tau \natural \zeta)}$$

[T-Data Parallel]
$$\frac{\Gamma \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1) \quad \Gamma \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)}{\Gamma \vdash D_1 \mid D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1 \cup \tau_2, \zeta_1 \sqcup \zeta_2)}$$

Figure 3.15: Typing rules for data trees.

Rule [T-Leaf Script] states that if a process respects the location policy, then it can be a leaf of a data tree that can stay at the location with this policy. The data tree $\mathtt{a}^{\tau}[\Box\Pi]$ has initial edge accessible to the processes with roles ρ such that $\tau \leq \rho$ and can be completely accessed by the processes with roles ρ such that $\tau \leq \rho$. This is recorded in the type where last two sets of roles are τ.

Rule [T-Leaf Pointer] states that a well-typed pointer can be a leaf of a data tree that can stay at a location with any policy. As in [T-Leaf Script] the last two sets of roles are the same as the set of roles of the edge.

Rule [T-Data Tree] states that an edge with roles τ can connect a parent node with a child node of type $\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta)$ only if $\tau \leqq \tau'$. This ensures that a process which can access the data tree D can also access the edge. Therefore, since in a well-typed tree the tree path $\mathtt{a}_1^{\tau_1}/\ldots/\mathtt{a}_n^{\tau_n}$ has the property $\tau_1 \leqq \ldots \leqq \tau_n$, we can reformulate accessibility of tree paths by processes as follows:

Lemma 3.3.1 (Access to data terms). *A process with roles can access a data term V in a well-typed data tree D if it can access the final edge of the tree path from the root of D to V.*

Proof. From [T-Data Tree], we know that in a well-typed tree the tree path $\mathtt{a}_1^{\tau_1}/\ldots/\mathtt{a}_n^{\tau_n}$ has the property $\tau_1 \leqq \ldots \leqq \tau_n$. Let ρ be the set of roles of the process. If $\tau_n \leq \rho$, then, by Lemma 3.2.13, $\tau_i \leq \rho$ for every $i \in \{1, \ldots, n-1\}$, which by Definition 3.2.7 means that the process with roles ρ has access to the data terms identified by the tree path. □

In the type of the conclusion of the rule [T-Data Tree] we use $\natural$ in order to obtain the set of roles that are allowed to access all the edges in a data tree.

Formally, we define

$$\tau \natural \zeta = \begin{cases} \tau & \text{if } \zeta = \{\top, \bot\} \\ \zeta & \text{otherwise.} \end{cases}$$

So, we get $\tau \natural \zeta = \tau$ if $D = \emptyset_D$ or if $\tau' = \zeta = \{\top, \bot\}$ (notice that in this case $\tau \leqq \tau'$ implies $\bot \in \tau$). Otherwise, $\tau \natural \zeta = \zeta$.

Rule [T-Data Parallel] states that the composition of two data trees has the union of their sets of initial roles as the set of initial roles. The composition of two data trees is completely accessible by the set obtained by joining the sets of roles which can completely access these data trees. From the definition of $\leqq$ we can easily show that $\sigma \leqq \tau_1 \cup \tau_2 \iff \sigma \leqq \tau_1 \wedge \sigma \leqq \tau_2$. This implies that the type of parallel composition of two data trees is well formed.

3.3.3 Typing processes and networks

Typing rules for channels and pure processes are given in Figure 3.16. In the following paragraphs we give their descriptions. Rule [T-Channel], as expected, states that channel $a^{T\mathsf{v}}$ can communicate values of the type $T\mathsf{v}$.

Rule [T-Inaction] states that the terminating process is well typed in any environment.

Rule [T-Pure Parallel] states that parallel composition of two processes is well typed in the same environment, for the same location policy and for same set of process roles as the component processes.

In Section 3.2.4, we have informally introduced characteristic roles for values. We now define characteristic roles for value types.

Definition 3.3.2 (Characteristic roles of value types). *The set of characteristic roles of value type $T\mathsf{v}$ (denoted $\mathcal{C}(T\mathsf{v})$) is defined as follows:*

$$\begin{array}{lcl} \mathcal{C}(\mathit{Channel}(T\mathsf{v})) & = & \mathcal{C}(T\mathsf{v}) \\ \mathcal{C}(\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)) & = & \tau \\ \mathcal{C}(\mathit{Path}(\alpha)) & = & \alpha \\ \mathcal{C}(\mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})) & = & \{\bot\} \\ \mathcal{C}(\mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})) & = & \{\bot\} \end{array}$$

We can now give the definition of characteristic roles for values and claim that these sets are appropriate formalizations of their descriptions.

Definition 3.3.3 (Characteristic roles of values). *If $\emptyset \vdash \mathsf{v} : T\mathsf{v}$, then $\mathcal{C}(\mathsf{v}) = \mathcal{C}(T\mathsf{v})$.*

Characteristic roles of a well-typed data tree correspond to the fourth set in its type. For the empty data tree it is $\{\top\}$. For a non-empty data tree, it is computed, by the typing rules for data trees, as the union of the set of roles assigned to the initial edges. Characteristic roles of a well-typed path correspond to the set computed by the typing rules for paths. It is straightforward to check that this set is actually the set of roles assigned to the final edge. The characteristic roles of the rest of the values are obviously the same as their informal descriptions.

Rule [T-Output] states that the process $u!\mathsf{v}$ is well typed in an environment, for the location policy $(\sigma, \mathcal{E}, \mathcal{D})$ and for the set of process roles ρ, if the channel u

[T-Channel]
$$\Gamma \vdash a^{T\mathsf{v}} : \mathit{Channel}(T\mathsf{v})$$

[T-Inaction]
$$\Gamma \vdash 0 : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$$

[T-Pure Parallel]
$$\frac{\Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad \Gamma \vdash Q : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}{\Gamma \vdash P \mid Q : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Replication]
$$\frac{\Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}{\Gamma \vdash *P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Output]
$$\frac{\Gamma \vdash u : \mathit{Channel}(T\mathsf{v}) \quad \Gamma \vdash \mathsf{v} : T\mathsf{v} \quad \mathcal{C}(T\mathsf{v}) \leq \rho}{\Gamma \vdash u!\mathsf{v} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Input]
$$\frac{\Gamma \vdash u : \mathit{Channel}(T\mathsf{v}) \quad \Gamma, x : T\mathsf{v} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad \mathcal{C}(T\mathsf{v}) \leq \rho}{\Gamma \vdash u?x.P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Go]
$$\frac{\Gamma \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D}) \quad \Gamma \vdash \lambda : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash \texttt{GO}\ \lambda.R : \mathit{Process}(\sigma', \mathcal{E}', \mathcal{D}', \rho)}$$

[T-Run]
$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \alpha \leq \rho}{\Gamma \vdash \texttt{RUN}_p : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Read]
$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \Gamma, \Gamma_\chi \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad \alpha \leq \rho}{\Gamma \vdash \texttt{READ}_p(\chi).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Change]
$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad \alpha \leq \rho \qquad \Gamma, \Gamma_\chi \vdash \begin{cases} \mathsf{V} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}) \text{ or} \\ \mathsf{V} : \mathit{Pointer}(\beta) \text{ or} \\ \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta') \quad \alpha \leqq \tau' \\ \text{if } \chi = x^{(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)} \text{ then } \zeta \leq \rho \end{cases}}{\Gamma \vdash \texttt{CHANGE}_p(\chi, \mathsf{V}).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Enable]
$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad (\rho, r) \in^+ \mathcal{E} \quad \alpha \leqq \{r\} \quad \alpha \leq \rho}{\Gamma \vdash \texttt{ENABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}$$

[T-Disable]
$$\frac{\Gamma \vdash p : \mathit{Path}(\alpha) \quad \Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho) \quad (\rho, r) \in^- \mathcal{D} \quad \alpha \leq \rho}{\Gamma \vdash \texttt{DISABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, r)}$$

Figure 3.16: Typing rules for channels and pure processes.

can transmit values of type $T\mathsf{v}$, which is the type of the value v, and if the set of characteristic roles of $T\mathsf{v}$ is accessible by the set of roles ρ.

Rule [T-Input] states that the process $u?x.P$ is well typed in Γ, for the location policy $(\sigma, \mathcal{E}, \mathcal{D})$ and for the set of process roles ρ, if the channel u can transmit values of type $T\mathsf{v}$ and if the set of characteristic roles of $T\mathsf{v}$ is accessible by the set of roles ρ. The continuation process P must be well typed in Γ extended with a type assignment which assigns $T\mathsf{v}$ to the variable x. This ensures that the variable x is correctly used in the continuation.

Rule [T-Replication] states that the replication of a process is well typed in an environment, for a policy and for a set of process roles, if the process is well typed in this environment, for that policy and for that set of process roles.

Rule [T-Run] states that process $\texttt{RUN}_p$, which aims to run embedded scripts at a location with policy $(\sigma, \mathcal{E}, \mathcal{D})$, is well typed in any environment, for a set of process roles ρ which are allowed to access the final edge of the path p. The same condition, $\alpha \leq \rho$, which ensures that processes look only for the data terms accessible to them (Lemmas 3.2.13 and 3.3.1), appears in premises of rules [T-Read], [T-Change], [T-Enable] and [T-Disable].

Rule [T-Go] states that the process $\texttt{GO}\ \lambda.R$ which aims to move to another location is well typed if the continuation R is well typed for the new location. This rule is applicable also in case current and new location are the same.

Typing rules [T-Read] and [T-Change] use additional type environment Γ_χ related to the patterns that appear in the corresponding processes. It assigns types to the variables of the pattern χ and it is defined as follows:

$$\Gamma_\chi = \begin{cases} x : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D}) & \text{if} \quad \chi = \Box x^{(\sigma,\mathcal{E},\mathcal{D})}, \\ x : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D}), y : \mathit{Path}(\alpha) & \text{if} \quad \chi = y^{\alpha} @ x^{(\sigma,\mathcal{E},\mathcal{D})}, \\ x : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta) & \text{if} \quad \chi = x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}. \end{cases}$$

The pattern variables are bound in the continuation P of the process $\texttt{READ}_p(\chi).P$ and in the data term V of the process $\texttt{CHANGE}_p(\chi, \mathsf{V})$. In rules [T-Read] and [T-Change] the premises which mention Γ_χ ensure that these variables are used according to their type annotations. The condition $\alpha \leqq \tau'$ in rule [T-Change] ensures that when we replace a data term by a data tree, the obtained data tree is well typed. In case we try to replace a subtree, i.e. if pattern is $\chi = x^{(\sigma,\mathcal{E},\mathcal{D},\tau,\zeta)}$, the condition $\zeta \leq \rho$ ensures that the whole subtree is accessible to the process.

Rules [T-Enable] states that the role r can be added to the data tree at a location with policy $(\sigma, \mathcal{E}, \mathcal{D})$ only if such action agrees with the policy $((\rho, r) \in^+ \mathcal{E})$. Furthermore, the premise $\alpha \leqq \{r\}$ ensures that the new role complies with the tree path and implicitly that the new data tree is well typed. This condition is not needed in rule [T-Disable], which ensures that role r is removed correctly with the respect the location policy. More precisely, rule [T-Disable] states that the role r can be removed from the data tree at a location with policy $(\sigma, \mathcal{E}, \mathcal{D})$ only if such action agrees with the policy $((\rho, r) \in^- \mathcal{D})$.

Typing rules for processes with roles are given in Figure 3.17. Rule [T-Role] explains connection of pure processes with processes with roles. Process with roles $P^{\neg\rho}$ is well typed for the location with policy $(\sigma, \mathcal{E}, \mathcal{D})$ if pure process P is well typed for the same location policy and roles ρ.

$$\text{[T-Role]}\quad \frac{\Gamma \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)}{\Gamma \vdash P^{\neg\rho} : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})} \qquad \text{[T-Restriction]}\quad \frac{\Gamma \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash (\nu a^{T_\mathsf{v}})R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}$$

$$\text{[T-Role Parallel]}\quad \frac{\Gamma \vdash R_1 : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D}) \quad \Gamma \vdash R_2 : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}{\Gamma \vdash R_1 \mid R_2 : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}$$

Figure 3.17: Typing rules for processes with roles.

$$\text{[T-Location]}\quad \frac{\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D}) \qquad \emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta) \quad \emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})}{\emptyset \vdash l[D \parallel R] : \mathit{Network}}$$

$$\text{[T-Network Restriction]}\quad \frac{\emptyset \vdash N : \mathit{Network}}{\emptyset \vdash (\nu a^{T_\mathsf{v}})N : \mathit{Network}}$$

$$\text{[T-Network Parallel]}\quad \frac{\emptyset \vdash N_1 : \mathit{Network} \quad \emptyset \vdash N_2 : \mathit{Network} \quad \mathcal{N}(N_1) \cap \mathcal{N}(N_2) = \emptyset}{\emptyset \vdash N_1 \mid N_2 : \mathit{Network}}$$

Figure 3.18: Typing rules for networks.

Rule [T-Restriction] states that process $(\nu a^{T_\mathsf{v}})R$ is well typed in an environment if process R is well typed in the same environment. Rule [T-Role Parallel] is similar to rule [T-Pure Parallel].

Finally, we give the typing rules for networks in Figure 3.18 and explain them in the following paragraphs. Rule [T-Location] states that a location $l[D \parallel R]$ with policy $(\sigma, \mathcal{E}, \mathcal{D})$ is well typed if the data tree D and the process with role R are well typed for the policy $(\sigma, \mathcal{E}, \mathcal{D})$.

The function $\mathcal{N}$ associates to a network the set of its location names:

$$\mathcal{N}(\mathbf{0}) = \emptyset \qquad \mathcal{N}(l[D \parallel R]) = \{l\} \qquad \mathcal{N}(N_1 \mid N_2) = \mathcal{N}(N_1) \cup \mathcal{N}(N_2).$$

It is used in [T-Network Parallel] to ensure that that each location name occurs at most once in a well-typed network. Rules [T-Network Restriction] and [T-Network Parallel] are similar to corresponding rules for processes.

A straightforward consequence of the type assignment rules are the following properties:

Proposition 3.3.4. *(i) Each location name occurs at most once in a well-typed network.*

(ii) If a location is well typed, then enclosed data tree and process with roles do not contain occurrences of free variables.

3.4 Type soundness

In the current section, we prove that the proposed type system guarantees the operational property of type preservation under reduction: all networks obtained by reduction starting from a well-typed network are again well typed. Then, we prove that the type system is safe: a well-typed network is well behaved. By proving these two properties, we conclude that our type system is sound.

Rules of the type system are syntax directed, meaning that a typing derivation of any term in the language has unique structure determined by its syntactic structure. The fact that the last typing rule applied in a typing derivation is uniquely determined allows us to use the reversal of the typing rules in the proofs.

Before we start with proofs of two main results we need to prove several auxiliary lemmas. The following two lemmas are standard and they say that typing is preserved by substitutions and by structural congruence.

Lemma 3.4.1 (Substitution).

1. *If* $\Gamma, x : T\mathsf{v} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ *and* $\Gamma \vdash \mathsf{v} : T\mathsf{v}$, *then* $\Gamma \vdash P\{\mathsf{v}/x\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$.
2. *If* $\Gamma, x : T\mathsf{v} \vdash \mathsf{V} : T$ *and* $\Gamma \vdash \mathsf{v} : T\mathsf{v}$, *then* $\Gamma \vdash \mathsf{V}\{\mathsf{v}/x\} : T$.
3. *If* $\Gamma_\chi \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ *and* $\mathsf{match}(\chi, \mathsf{V})$, *then* $\emptyset \vdash P\{\!\{\mathsf{V}/|\chi|\}\!\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$.
4. *If* $\Gamma_\chi \vdash \mathsf{V} : T$ *and* $\mathsf{match}(\chi, \mathsf{W})$, *then* $\emptyset \vdash \mathsf{V}\{\!\{\mathsf{W}/|\chi|\}\!\} : T$.

Proof.

1. If x does not occur free in P, then $P\{\mathsf{v}/x\} = P$. In that case $\Gamma \vdash P\{\mathsf{v}/x\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ trivially holds. We assume $x \in \mathsf{fv}(P)$ and proceed by induction on the derivation of $\Gamma, x : T\mathsf{v} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We show few interesting cases.

   [T-Pure Parallel] In this case we know that $P = P_1 \mid P_2$ and $\Gamma, x : T\mathsf{v} \vdash P_1 : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and $\Gamma, x : T\mathsf{v} \vdash P_2 : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We derive the proof by induction hypothesis and rule [T-Pure Parallel].

   [T-Input] In this case $P = u?y.Q$ and $\Gamma, x : T\mathsf{v} \vdash u : \mathit{Channel}(T\mathsf{w})$ and $\Gamma, x : T\mathsf{v}, y : T\mathsf{w} \vdash Q : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and $\mathcal{C}(T\mathsf{w}) \leq \rho$. The only interesting case is when $u = x$. Then, $P = x?y.Q$ and $T\mathsf{v} = \mathit{Channel}(T\mathsf{w})$ and $P\{\mathsf{v}/x\} = \mathsf{v}?y.Q\{\mathsf{v}/x\}$. By induction hypothesis we obtain $\Gamma, y : T\mathsf{w} \vdash Q\{\mathsf{v}/x\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and conclude the proof by an application of [T-Input].

   [T-Go] In this case $P = \texttt{GO}\ \lambda.R$ and $\Gamma, x : T\mathsf{v} \vdash R : \mathit{RoleProcess}(\sigma', \mathcal{E}', \mathcal{D}')$ and $\Gamma, x : T\mathsf{v} \vdash \lambda : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$. By induction on the structure of

R, we prove $\Gamma \vdash R\{\mathsf{v}/x\} : \mathit{RoleProcess}(\sigma', \mathcal{E}', \mathcal{D}')$. We distinguish two cases, $x = \lambda$ and $x \neq \lambda$. If $x = \lambda$, then, from [T-Value Variable], $T\mathsf{v} = \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$, so we know $\Gamma \vdash \mathsf{v} : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$. We conclude the proof by [T-Go]. If $x \neq \lambda$, then $\Gamma \vdash \lambda : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$, and we conclude the proof by [T-Go].

[T-Run] In this case $P = \texttt{RUN}_p$ and $\Gamma, x : T\mathsf{v} \vdash p : \mathit{Path}(\alpha)$ and $\alpha \leq \rho$. The most interesting case is when $p = q/x$. For notational simplicity we assume $x \notin \mathsf{fv}(q)$. From $\Gamma, x : T\mathsf{v} \vdash p : \mathit{Path}(\alpha)$ and [T-Path Composition] we obtain $\Gamma, x : T\mathsf{v} \vdash q : \mathit{Path}(\beta)$ and $\Gamma, x : T\mathsf{v} \vdash x : \mathit{Path}(\alpha)$. So, from [T-Value Variable] we get $T\mathsf{v} = \mathit{Path}(\alpha)$ and from $x \notin \mathsf{fv}(q)$ we obtain $\Gamma \vdash q : \mathit{Path}(\beta)$. We derive the proof of $\Gamma \vdash \texttt{RUN}_{q/\mathsf{v}}$ by [T-Path Composition] and [T-Run].

In cases when the last applied rules in the derivation of $\Gamma, x : T\mathsf{v} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ are [T-Read], [T-Change], [T-Enable] or [T-Disable], the proof is derived using the same reasoning on paths as in the case of rule [T-Run] and induction hypothesis. In addition, case [T-Change] uses 3.4.1.(2).

2. We distinguish cases based on V.

   (a) $\mathsf{V} = \Box\Pi$. The only interesting case is when $\Pi = x$. Then, $T = T\mathsf{v} = \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$. From [T-Script] we know that $\mathsf{v} = \Box\Pi'$, so $\mathsf{V}\{\mathsf{v}/x\} = \Box\Pi'$ and the proof is trivial.

   (b) $\mathsf{V} = p@\lambda$. This case is similar to (a).

   (c) $\mathsf{V} = D$. If x does not occur free in D, then $D\{\mathsf{v}/x\} = D$. In that case $\Gamma \vdash D\{\mathsf{v}/x\} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ trivially holds. We assume $x \in \mathsf{fv}(D)$ and proceed by induction on the derivation of $\Gamma, x : T\mathsf{v} \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$.

   [T-Empty Data] Then $D = \emptyset_D$. Since $D\{\!\{\mathsf{V}/|\chi|\}\!\} = \emptyset_D = D$, we derive the proof trivially.

   [T-Leaf Script] Then $D = \mathsf{a}^\tau[\Box\Pi]$ and $\Gamma_\chi \vdash \Box\Pi : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$. From [T-Script] we know that $\Gamma_\chi \vdash \Pi : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$. The only interesting possibility is when $\Pi = x$ and $\chi = \Box x^{(\sigma, \mathcal{E}, \mathcal{D})}$. From $\mathsf{match}(\chi, \mathsf{V})$ we know that $\mathsf{V} = \Box R$ and $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$. Then $D\{\!\{\mathsf{V}/|\chi|\}\!\} = \mathsf{a}^\tau[\Box R]$. We conclude the proof by [T-Script] and [T-Leaf Script].

   [T-Leaf Pointer] Then $D = \mathsf{a}^\tau[p@\lambda]$ and $\Gamma_\chi \vdash p@\lambda : \mathit{Pointer}(\alpha)$. From [T-Pointer] we know that $\Gamma_\chi \vdash p : \mathit{Path}(\alpha)$ and $\Gamma_\chi \vdash \lambda : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$. The most interesting possibility is when $p = y$ and $\lambda = x$ and $\chi = y^\alpha@x^{(\sigma, \mathcal{E}, \mathcal{D})}$. From $\mathsf{match}(\chi, \mathsf{V})$ we know that $\mathsf{V} = q@l$ and $\emptyset \vdash q : \mathit{Path}(\alpha)$ and $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$. Then $D\{\!\{\mathsf{V}/|\chi|\}\!\} = \mathsf{a}^\tau[q@l]$. We conclude the proof by [T-Pointer] and [T-Leaf Pointer].

[T-Data Tree] Then $D = \mathsf{a}^{\tau}[D']$ and for τ' and ζ' such that $\tau \leqq \tau'$ and $\zeta = \tau \natural \zeta'$ it holds $\Gamma_{\chi} \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$. By induction hypothesis we obtain $\emptyset \vdash D'\{\!\{\mathsf{V}/|\chi|\}\!\} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$. We conclude the proof by [T-Data Tree].

[T-Data Parallel] In this case we know that $D = D_1 \mid D_2$ and $\Gamma_{\chi} \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\Gamma_{\chi} \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$ and $\tau = \tau_1 \cup \tau_2$ and $\zeta = \zeta_1 \sqcup \zeta_2$. The proof is derived by induction hypothesis and [T-Data Parallel].

3. If pattern variables do not occur free in P, then $P\{\!\{\mathsf{V}/|\chi|\}\!\} = P$. In that case $\emptyset \vdash P\{\!\{\mathsf{V}/|\chi|\}\!\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ trivially holds. We assume pattern variables are free variables of P and proceed by induction on the derivation of $\Gamma_{\chi} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We show few interesting cases.

   [T-Go] In this case $P = \texttt{GO}\ \lambda.R$ and $\Gamma_{\chi} \vdash R : \mathit{RoleProcess}(\sigma', \mathcal{E}', \mathcal{D}')$ and $\Gamma_{\chi} \vdash \lambda : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$. By induction on the structure of R, we prove $\emptyset \vdash R\{\!\{\mathsf{V}/|\chi|\}\!\} : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$. We derive the proof in case $\lambda = x$ and $\chi = y^{\alpha}@x^{(\sigma', \mathcal{E}', \mathcal{D}')}$. From $\mathsf{match}(\chi, \mathsf{V})$ we know that $\mathsf{V} = p@l$ and $\emptyset \vdash p : \mathit{Path}(\alpha)$ and $\emptyset \vdash l : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$. Since, $P\{\!\{\mathsf{V}/|\chi|\}\!\} = \texttt{GO}\ l.R$ we conclude the proof by [T-Go].

   [T-Enable] In this case $P = \texttt{ENABLE}_p(r).Q$. From [T-Enable] we know that $\Gamma_{\chi} \vdash p : \mathit{Path}(\alpha)$ and $\Gamma_{\chi} \vdash Q : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We derive the proof in case when $\chi = y^{\alpha}@x^{(\sigma', \mathcal{E}', \mathcal{D}')}$ and $y \in \mathsf{fv}(p)$. For notational simplicity we assume $p = q/y$ and $y \notin \mathsf{fv}(q)$. From $\Gamma_{\chi} \vdash p : \mathit{Path}(\alpha)$ and [T-Path Composition] we obtain $\Gamma_{\chi} \vdash q : \mathit{Path}(\beta)$ and $\Gamma_{\chi} \vdash y : \mathit{Path}(\alpha)$. From $\mathsf{match}(\chi, \mathsf{V})$ we know that $\mathsf{V} = p'@l$ and $\emptyset \vdash p' : \mathit{Path}(\alpha)$ and $\emptyset \vdash l : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$. From $x \notin \mathsf{fv}(q)$ we obtain $\Gamma \vdash q : \mathit{Path}(\beta)$. We derive the proof of $\Gamma \vdash \texttt{ENABLE}_{q/p'}(r).Q\{\!\{\mathsf{V}/|\chi|\}\!\}$ by [T-Path Composition], induction hypothesis and [T-Enable].

   [T-Change] In this case $P = \texttt{CHANGE}_p(\chi', \mathsf{W}).Q$. Interesting cases are when variables of the pattern χ are free variables of p and free variables of W. In first case we use same reasoning as in the case of [T-Enable] and in the second we use Lemma 3.4.1(4).

4. The most interesting case is when $\mathsf{V} = D$. Then $T = \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$. If pattern variables do not occur free in D, then $D\{\!\{\mathsf{W}/|\chi|\}\!\} = D$. In that case $\emptyset \vdash D\{\!\{\mathsf{W}/|\chi|\}\!\} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ trivially holds. We assume pattern variables are free variables of D and proceed by induction on the derivation of $\Gamma_{\chi} \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$.

   [T-Empty Data] In this case $D = \emptyset_D$. Since $D\{\!\{\mathsf{W}/|\chi|\}\!\} = \emptyset_D = D$, we derive the proof trivially.

   [T-Leaf Script] In this case $D = \mathsf{a}^{\tau}[\Box\Pi]$ and $\Gamma_{\chi} \vdash \Box\Pi : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$. From [T-Script] we know that $\Gamma_{\chi} \vdash \Pi : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$. The most

interesting possibility is when $\Pi = x$ and $\chi = \Box x^{(\sigma,\mathcal{E},\mathcal{D})}$. From $\mathsf{match}(\chi, \mathsf{W})$ we know that $\mathsf{W} = \Box R$ and $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$. Then $D\{\!\{\mathsf{W}/|\chi|\}\!\} = \mathsf{a}^{\tau}[\Box R]$. We conclude the proof by [T-Script] and [T-Leaf Script].

[T-Leaf Pointer] In this case $D = \mathsf{a}^{\tau}[p@\lambda]$ and $\Gamma_\chi \vdash p@\lambda : \mathit{Pointer}(\alpha)$. From [T-Pointer] we know that $\Gamma_\chi \vdash p : \mathit{Path}(\alpha)$ and that $\Gamma_\chi \vdash \lambda : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$. The most interesting possibility is when $\chi = y^{\alpha}@x^{(\sigma,\mathcal{E},\mathcal{D})}$ and $p = y$ and $\lambda = x$. From $\mathsf{match}(\chi, \mathsf{W})$ we know that $\mathsf{W} = q@l$ and $\emptyset \vdash q : \mathit{Path}(\alpha)$ and $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$. Then $D\{\!\{\mathsf{W}/|\chi|\}\!\} = \mathsf{a}^{\tau}[q@l]$. We conclude the proof by [T-Pointer] and [T-Leaf Pointer].

[T-Data Tree] In this case $D = \mathsf{a}^{\tau}[D']$ and for τ' and ζ' such that $\tau \leqq \tau'$ and $\zeta = \tau \natural \zeta'$ it holds $\Gamma_\chi \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$. By induction hypothesis we obtain $\emptyset \vdash D'\{\!\{\mathsf{W}/|\chi|\}\!\} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$. We conclude the proof by [T-Data Tree].

[T-Data Parallel] In this case we know that $D = D_1 \mid D_2$ and $\Gamma_\chi \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\Gamma_\chi \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$ and $\tau = \tau_1 \cup \tau_2$ and $\zeta = \zeta_1 \sqcup \zeta_2$. The proof is derived by induction hypothesis and [T-Data Parallel].

□

Lemma 3.4.2.

1. *If* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ *and* $D \equiv D'$, *then* $\emptyset \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$;
2. *If* $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ *and* $R \equiv R'$, *then* $\emptyset \vdash R' : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$;
3. *If* $\emptyset \vdash N : \mathit{Network}$ *and* $N \equiv N'$, *then* $\emptyset \vdash N' : \mathit{Network}$.

Proof. Straightforward, by case analysis on the derivation of $D \equiv D'$, $R \equiv R'$ and $N \equiv N'$. □

The next lemma shows that if a data tree is well typed, then the union of the annotations on its initial edges equals the fourth set of its type, i.e. it equals the characteristics roles of the tree.

Lemma 3.4.3. *If* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ *and* $D \neq \emptyset_D$, *then* τ *is the union of the annotations on the initial edges of* D.

Proof. The proof is by induction on the structure of D.

$D = \mathsf{a}^{\tau'}[\mathsf{V}]$ From the typing rules for data tree, we know that in this case $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ must have been derived by one of rules [T-Leaf Script], [T-Leaf Pointer] or [T-Data Tree]. In any of the possible cases it holds $\tau' = \tau$.

$D = D_1 \mid D_2$ In this case, we know that $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ must have been derived by from [T-Data Parallel]. So, we know that $\Gamma \vdash D_1 :$

$\mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\Gamma \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, where $\tau_1 \cup \tau_2 = \tau$ and $\zeta_1 \sqcup \zeta_2 = \zeta$. We derive the proof by induction hypothesis and rule [T-Data Parallel]. □

From this lemma we know that if $\emptyset \vdash \mathtt{a}^{\tau'}[\mathsf{V}] : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$, then $\tau' = \tau$. We use this fact in proofs whenever we derive the proof by induction on the structure of D.

The next lemma shows that if a path is well typed, then the annotations on its final edge equal the set of roles in the type, i.e. it equals the set of characteristic roles of the path.

Lemma 3.4.4. *If* $\Gamma \vdash \mathtt{a}^{\beta} : \mathit{Path}(\alpha)$, *then* $\beta = \alpha$. *If* $\Gamma \vdash p/q : \mathit{Path}(\alpha)$, *then* $\Gamma \vdash q : \mathit{Path}(\alpha)$.

Proof. Straightforward from [T-Path Edge] and [T-Path Composition]. □

We use the first statement of this lemma in order to simplify notation in proofs whenever we derive the proof by cases or induction on the structure of the path.

The next lemma shows that scripts in a well-typed data tree are well typed for the same location policy as the data tree, which implies that they can be safely activated.

Lemma 3.4.5. *Let* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$. *If* $R \in \mathtt{run}(D, p)$, *then* $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, by the definition of function run, given in Figure 3.10, $\mathtt{run}(D, p) = \emptyset$, so $R \in \mathtt{run}(D, p)$ does not hold.

$\boxed{D = \mathtt{b}^{\tau}[\Box\Pi]}$ We have two cases:

(i) $p = \mathtt{b}^{\alpha}$ and $\tau \leqq \alpha$. In this case, $\Pi \in \mathtt{run}(D, p)$. From [T-Leaf Script] and [T-Script] we obtain $\emptyset \vdash \Pi : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

(ii) $p \neq \mathtt{b}^{\alpha}$ or ($p = \mathtt{b}^{\alpha}$ and $\tau \not\leqq \alpha$). In this case $\mathtt{run}(D, p) = \emptyset$, so $R \in \mathtt{run}(D, p)$ does not hold.

$\boxed{D = \mathtt{b}^{\tau}[p@\lambda]}$ In this case $\mathtt{run}(D, p) = \emptyset$, so $R \in \mathtt{run}(D, p)$ does not hold.

$\boxed{D = \mathtt{b}^{\tau}[D']}$ From [T-Data Tree] we have that $\Gamma \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$, for some τ' and ζ' (such that $\tau \leqq \tau'$ and $\zeta = \tau \natural \zeta'$). We derive the proof by induction on structure of p.

$\boxed{p = \mathtt{a}^{\alpha}}$ In this case $\mathtt{run}(D, p) = \emptyset$, so $R \in \mathtt{run}(D, p)$ does not hold.

$\boxed{p = \mathtt{a}^{\alpha}/q}$ We have two cases:

(i) $\mathtt{a} \neq \mathtt{b}$ or ($\mathtt{a} = \mathtt{b}$ and $\tau \not\leqq \alpha$). In this case $\mathtt{run}(D, p) = \emptyset$, so $R \in \mathtt{run}(D, p) \neq \emptyset$ does not hold.

(ii) $\mathtt{a} = \mathtt{b}$ and $\tau \leqq \alpha$. In this case $\mathtt{run}(D, p) = \mathtt{run}(D', q)$. By induction hypothesis, we conclude the proof.

$\boxed{D = D_1 \mid D_2}$ In this case by the definition of function $\mathtt{run}$, we know that $R \in \mathtt{run}(D_1, p)$ or $R \in \mathtt{run}(D_2, p)$.
Since [T-Data Parallel] implies $\emptyset \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\emptyset \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, we conclude $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ by induction hypothesis.

□

The next lemma shows that the data terms in substitutions obtained by the function $\mathtt{read}$ are matching the required pattern.

Lemma 3.4.6. *Let* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$. *If* $\{\!\{\mathsf{V}/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$, *then* $\mathsf{match}(\chi, \mathsf{V})$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ By the definition of function $\mathtt{read}$, given in Figure 3.10, $\mathtt{read}(D, p, \chi) = \emptyset$, so $\{\!\{\mathsf{V}/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$ does not hold.

$\boxed{D = \mathtt{b}^\tau[\Box\Pi]}$ We have two cases:

(i) $p = \mathtt{b}^\alpha$ and $\tau \leqq \alpha$. By the definition of function $\mathtt{read}$, we know that if $\{\!\{\Box\Pi/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$, then $\mathsf{match}(\chi, \Box\Pi)$.

(ii) $p \neq \mathtt{b}^\alpha$ or ($p = \mathtt{b}^\alpha$ and $\tau \not\leqq \alpha$). In this case $\mathtt{read}(D, p, \chi) = \emptyset$, so $\{\!\{\Box\Pi/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$ does not hold.

$\boxed{D = \mathtt{b}^\tau[p@\lambda]}$ Similar to the previous case.

$\boxed{D = \mathtt{b}^\tau[D']}$ From [T-Data Tree] we have that $\Gamma \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$, for some τ' and ζ' (such that $\tau \leqq \tau'$ and $\zeta = \tau \natural \zeta'$). We derive the proof by induction on structure of p.

$\boxed{p = \mathtt{a}^\alpha}$ By the definition of function $\mathtt{read}$, we know that if $\{\!\{D'/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$, then $\mathsf{match}(\chi, D')$.

$\boxed{p = \mathtt{a}^\alpha/q}$ We have two cases:

(i) $\mathtt{a} \neq \mathtt{b}$ or ($\mathtt{a} = \mathtt{b}$ and $\tau \not\leqq \alpha$). In this case $\mathtt{read}(D, p, \chi) = \emptyset$, so $\{\!\{\Box\Pi/|\chi|\}\!\} \in \mathtt{read}(D, p, \chi)$ does not hold.

(ii) $\mathtt{a} = \mathtt{b}$ and $\tau \leqq \alpha$. In this case $\mathtt{read}(D, p, \chi) = \mathtt{read}(D', q, \chi)$. By induction hypothesis, we conclude the proof.

$\boxed{D = D_1 \mid D_2}$ By the definition of function $\mathtt{read}$, we know that $\{\!\{\mathsf{V}/|\chi|\}\!\} \in \mathtt{read}(D_1, p, \chi)$ or $\{\!\{\mathsf{V}/|\chi|\}\!\} \in \mathtt{read}(D_2, p, \chi)$. Since [T-Data Parallel] implies $\Gamma \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\Gamma \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, we derive the proof by induction hypothesis. □

The following lemma shows that if a well-typed data tree is changed by a well-typed process the resulting data tree is still well typed. The functions change, is defined in such way that is cannot affect the initial edges of the tree. For this reason characteristic roles of the data tree do not change.

Lemma 3.4.7. *Let* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ *and* $\emptyset \vdash \mathtt{CHANGE}_p(\chi, \mathsf{V}).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. *If* $E = \mathtt{change}(D, p, \chi, \mathsf{V})$, *then there exists* ζ' *such that* $\emptyset \vdash E : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta')$.

Proof. [T-Change] implies

- (H.1) $\emptyset \vdash p : \mathit{Path}(\alpha)$
- (H.2) $\Gamma_\chi \vdash \begin{cases} \mathsf{V} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}) \text{ or} \\ \mathsf{V} : \mathit{Pointer}(\beta) \text{ or} \\ \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'', \zeta'') \quad \alpha \leqq \tau'' \end{cases}$

The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, by the definition of function change, given in Figure 3.10, $E = \mathtt{change}(D, p, \chi, \mathsf{V}) = \emptyset_D = D$, so the proof is trivial.

$\boxed{D = \mathtt{b}^\tau[\Box\Pi]}$ We have two cases:

(i) $p = \mathtt{b}^\alpha$ and $\tau \leqq \alpha$ and $\mathsf{match}(\Box\Pi, \chi)$. By the definition of function change, we know that $E = \mathtt{b}^\tau[\mathsf{V}\{\!\{\Box\Pi/|\chi|\}\!\}]$. Lemma 3.4.1 implies $\mathsf{V}\{\!\{\Box\Pi/|\chi|\}\!\}$ has the same type as V, i.e.

$$\emptyset \vdash \begin{cases} \mathsf{V}\{\!\{\Box\Pi/|\chi|\}\!\} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}) \text{ or} \\ \mathsf{V}\{\!\{\Box\Pi/|\chi|\}\!\} : \mathit{Pointer}(\beta) \text{ or} \\ \mathsf{V}\{\!\{\Box\Pi/|\chi|\}\!\} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'', \zeta'') \quad \alpha \leqq \tau''. \end{cases}$$

If V is a script or a pointer, the proof is straightforward. If V is a data tree, from $\tau \leqq \alpha$ and $\alpha \leqq \tau''$, by transitivity of $\leqq$, we obtain $\tau \leqq \tau''$. We conclude the proof in this case by an application of [T-Data Tree].

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{D = \mathtt{b}^\tau[p@\lambda]}$ Similar to the previous case.

$\boxed{D = \mathtt{b}^\tau[D']}$ From [T-Data Tree] we have that $\emptyset \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau''', \zeta''')$, for some τ''' and ζ''' (such that $\tau \leqq \tau'''$ and $\zeta = \tau \natural \zeta'''$). We derive the proof by induction on structure of p.

$\boxed{p = \mathtt{a}^\alpha}$ We have two cases:

(i) $\mathtt{a} = \mathtt{b}$ and $\tau \leqq \alpha$ and $\mathsf{match}(\chi, D')$. In this case $E = \mathtt{b}^\tau[\mathsf{V}\{\!\{D'/\chi\}\!\}]$ Lemma 3.4.1 implies $\mathsf{V}\{\!\{D'/\chi\}\!\}$ has the same type as V. We derive the proof by an application of the typing rule for data trees which corresponds to V.

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{p = \mathsf{a}^{\beta}/q}$ (H.1) and [T-Path Composition] imply $\emptyset \vdash q : \mathit{Path}(\alpha)$. We have two cases:

(i) $\mathsf{a} = \mathsf{b}$ and $\tau \leqq \beta$. In this case $E = \mathsf{b}^{\tau}[\mathtt{change}(D', q, \chi, \mathsf{V})]$. By induction hypothesis, we obtain $\emptyset \vdash \mathtt{change}(D', q, \chi, \mathsf{V}) : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau''', \zeta^{iv})$. Since $\tau \leqq \tau'''$, we can conclude the proof by an application of[T-Data Tree].

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{D = D_1 \mid D_2}$ In this case, from the definition of function change, we know that $E = E_1 \mid E_2$, where $E_1 = \mathtt{change}(D_1, p, \chi, \mathsf{V})$ and $E_2 = \mathtt{change}(D_2, p, \chi, \mathsf{V})$. [T-Data Parallel] implies $\Gamma \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\Gamma \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, where $\tau_1 \cup \tau_2 = \tau$ and $\zeta_1 \sqcup \zeta_2 = \zeta$. By induction hypothesis, we conclude $\Gamma \vdash E_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1')$ and $\Gamma \vdash E_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2')$. From the definition of $\leqq$ we know that $\sigma \leqq \tau_1 \wedge \sigma \leqq \tau_2 \implies \sigma \leqq \tau_1 \cup \tau_2$. We conclude the proof with an application of rule [T-Data Parallel]. □

The following two lemmas give us the types of data terms on which roles have been enabled or disabled. From the definitions of the functions $^{+r}$ and $^{-r}$, given in the Figure 3.10, we see that scripts and pointers are not affected so their types do not change. If the argument is a data tree, function $^{+r}$ adds role r to its initial edges and function $^{-r}$ removes this role from all the edges. These changes are shown in the types. The role must be removed from all the edges in order for the data tree to stay well typed and consequently for the access to data not to be discontinued.

Lemma 3.4.8.

1. *If* $\emptyset \vdash \mathsf{V} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$, *then* $\emptyset \vdash (\mathsf{V})^{+r} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$;
2. *If* $\emptyset \vdash \mathsf{V} : \mathit{Pointer}(\alpha)$, *then* $\emptyset \vdash (\mathsf{V})^{+r} : \mathit{Pointer}(\alpha)$;
3. *If* $\emptyset \vdash \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$, *then* $\emptyset \vdash (\mathsf{V})^{+r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$ *and if* $\mathsf{V} \neq \emptyset_D$, *then* $\tau' = \tau \cup \{r\}$.

Proof. We only prove item (3), since items (1) and (2) are trivial. From $\emptyset \vdash \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$, we know that V is a data tree, i.e. $\mathsf{V} = D$. The proof is by induction on the structure of D.

$\boxed{[D = \emptyset_D]}$ Then $(D)^{+r} = \emptyset_D$ and so, by [T-Empty Data] we get $\emptyset \vdash (\mathsf{V})^{+r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$, where $\tau' = \tau = \{\top\}$.

$\boxed{D = \mathsf{b}^{\tau}[\mathsf{W}]}$ In this case $(D)^{+r} = \mathsf{b}^{\tau \cup \{r\}}[\mathsf{W}]$. Furthermore, from the typing rules for data trees, we know

$$\emptyset \vdash \begin{cases} \mathsf{W} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}), \text{ or} \\ \mathsf{W} : \mathit{Pointer}(\beta), \text{ or} \\ \mathsf{W} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'', \zeta'') \text{ and } \tau \leqq \tau'' \end{cases}$$

In case W is a script or a pointer, we conclude the proof with an application of [T-Leaf script] or [T-Leaf Pointer], respectively. From $\tau \leqq \tau''$, by the definition of $\leqq$, we get that $\tau \cup \{r\} \leqq \tau''$. So, in case W is a data tree, we can derive the proof with an application of [T-Data Tree].

$\boxed{D = D_1 \mid D_2}$ In this case by the definition of function $^{+r}$, given in Figure 3.10, we know that $(D_1 \mid D_2)^{+r} = (D_1)^{+r} \mid (D_2)^{+r}$. Since [T-Data Parallel] implies $\emptyset \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\emptyset \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, we derive the proof by induction hypothesis.

□

Lemma 3.4.9.

1. *If* $\emptyset \vdash \mathsf{V} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$, *then* $\emptyset \vdash (\mathsf{V})^{-r} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D})$;
2. *If* $\emptyset \vdash \mathsf{V} : \mathit{Pointer}(\alpha)$, *then* $\emptyset \vdash (\mathsf{V})^{-r} : \mathit{Pointer}(\alpha)$;
3. *If* $\emptyset \vdash \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$, *then* $\emptyset \vdash (\mathsf{V})^{-r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$ *and* $\tau' = \tau \setminus\setminus r$.

Proof. Regarding item (1), from [T-Script], we know that V is a script. So, item (1) trivially holds because in these cases we know that $(\Box\Pi)^{-r} = \Box\Pi$. Similarly, item (2) also holds. We only prove item (3). From $\emptyset \vdash \mathsf{V} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$ we know that V is a data tree, i.e. $\mathsf{V} = D$. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ Then $(D)^{-r} = \emptyset_D$ and so, by [T-Empty Data] we get $\emptyset \vdash (\mathsf{V})^{-r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$ where $\tau' = \tau \setminus\setminus r = \tau = \{\top\}$ and $\zeta' = \zeta = \{\bot, \top\}$.

$\boxed{D = \mathsf{b}^{\tau}[\mathsf{W}]}$ In this case $(D)^{-r} = \mathsf{b}^{\tau \setminus\setminus r}[(\mathsf{W})^{-r}]$. Furthermore, we know

$$\emptyset \vdash \begin{cases} \mathsf{W} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}), \text{ or} \\ \mathsf{W} : \mathit{Pointer}(\beta), \text{ or} \\ \mathsf{W} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'', \zeta'') \text{ and } \tau \leqq \tau'' \end{cases}$$

In all three cases we can apply the induction hypothesis and get

$$\emptyset \vdash \begin{cases} (\mathsf{W})^{-r} : \mathit{Script}(\sigma, \mathcal{E}, \mathcal{D}), \text{ or} \\ (\mathsf{W})^{-r} : \mathit{Pointer}(\beta), \text{ or} \\ (\mathsf{W})^{-r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau''', \zeta''') \text{ and } \tau''' = \tau'' \setminus\setminus r \end{cases}$$

In case W is a script or a pointer, we know that $(\mathsf{W})^{-r} = \mathsf{W}$. We also know that it holds $\sigma \leqq \tau \implies \sigma \leqq \tau \setminus\setminus r$. So, in these cases, we derive the proof by an application of [T-Leaf script] or [T-Leaf Pointer], respectively. From $\tau \leqq \tau''$, by the definition of $\leqq$ and $\setminus\setminus$, we get that $\tau \setminus\setminus r \leqq \tau'' \setminus\setminus r$. So, in case W is a data tree, we can derive the proof with an application of [T-Data Tree].

$\boxed{D = D_1 \mid D_2}$ In this case by the definition of function $^{-r}$, given in Figure 3.10, we know that $(D_1 \mid D_2)^{-r} = (D_1)^{-r} \mid (D_2)^{-r}$. Since [T-Data Parallel] implies $\emptyset \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\emptyset \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, we derive the proof by induction hypothesis. Notice that it holds $(\tau_1 \setminus\setminus r) \cup (\tau_2 \setminus\setminus r) = (\tau_1 \cup \tau_2) \setminus\setminus r$. □

The next lemma shows that if a well-typed process updates roles on a well-typed data tree, the resulting data tree is also well typed. Characteristic roles are not affected by functions `enable` and `disable`.

Lemma 3.4.10. *Let* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$. *If*

1. $\emptyset \vdash \mathtt{ENABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ *and* $E = \mathtt{enable}(D, p, r)$, *or*
2. $\emptyset \vdash \mathtt{DISABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ *and* $E = \mathtt{disable}(D, p, r)$,

then $\emptyset \vdash E : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta')$.

Proof. 1. The proof is by induction on the structure of D and p. [T-Enable] implies (H.1) $\Gamma \vdash p : \mathit{Path}(\alpha)$ and $(\rho, r) \in^+ \mathcal{E}$ and $\alpha \leqq \{r\}$ and $\alpha \leq \rho$.

$\boxed{D = \emptyset_D}$ In this case, from [T-Empty Data], we know $\tau = \{\top\}$ and $\zeta = \{\bot, \top\}$. From definition of function `enable`, given in Figure 3.10, we know that $E = \emptyset_D$. We derive the proof by [T-Empty Data] with $\tau' = \tau = \{\top\}$ and $\zeta' = \zeta = \{\bot, \top\}$.

$\boxed{D = \mathtt{b}^\tau[\Box\Pi]}$ We have two cases:

(i) $p = \mathtt{b}^\alpha$ and $\tau \leqq \alpha$. By the definition of functions `enable` and $^{+r}$, we know that $E = \mathtt{b}^\tau[\Box\Pi^{+r}] = \mathtt{b}^\tau[\Box\Pi] = D$. So, the proof is trivial.

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{D = \mathtt{b}^\tau[p@\lambda]}$ Similar to the previous case.

$\boxed{D = \mathtt{b}^\tau[D']}$ In this case, from [T-Data Tree] we know that $\emptyset \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'', \zeta'')$, for some τ'' and ζ'' (such that $\tau \leqq \tau''$ and $\zeta = \tau \natural \zeta''$). We derive the proof by induction on structure of p.

$\boxed{p = \mathtt{a}^\alpha}$ We have two cases:

(i) $\mathtt{a} = \mathtt{b}$ and $\tau \leqq \alpha$. In this case $E = \mathtt{b}^\tau[D'^{+r}]$. Lemma 3.4.8 implies $\emptyset \vdash D'^{+r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'' \cup \{r\}, \zeta''')$ or that, in case $D' = \emptyset_D$, $\emptyset \vdash D'^{+r} : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \{\top\}, \zeta''')$ We derive the proof by an application of [T-Data Tree].

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{p = \mathtt{a}^\beta/q}$ (H.1) and [T-Path Composition] imply $\Gamma \vdash q : \mathit{Path}(\alpha)$. We have two cases:

(i) $\mathtt{a} = \mathtt{b}$ and $\tau \leqq \beta$. In this case $E = \mathtt{b}^\tau[\mathtt{enable}(D', q, r)]$. After we apply the induction hypothesis, we obtain $\emptyset \vdash \mathtt{enable}(D', q, r) : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau'' \cup \{r\}, \zeta^{iv})$. Since $\tau \leqq \tau'' \cup \{r\}$, we can conclude the proof by an application of [T-Data Tree].

(ii) otherwise, $E = D$, so the proof is trivial.

$\boxed{D = D_1 \mid D_2}$ In this case, from the definition of function `enable`, we know that $E = E_1 \mid E_2$, where $E_1 = \texttt{enable}(D_1, p, r)$ and $E_2 = \texttt{enable}(D_2, p, r)$. [T-Data Parallel] implies $\emptyset \vdash D_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1, \zeta_1)$ and $\emptyset \vdash D_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2, \zeta_2)$, where $\tau_1 \cup \tau_2 = \tau$ and $\zeta_1 \sqcup \zeta_2 = \zeta$. By induction hypothesis we conclude $\emptyset \vdash E_1 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_1', \zeta_1')$ and $\emptyset \vdash E_2 : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau_2', \zeta_2')$. From the definition of $\leqq$ we know that $\sigma \leqq \tau_1' \wedge \sigma \leqq \tau_2' \implies \sigma \leqq \tau_1' \cup \tau_2'$. We conclude the proof with an application of rule [T-Data Parallel].

2. Similar to the previous case.

□

Some, but not all, of the previous lemmas hold also for terms well typed in an environment $\Gamma \neq \emptyset$. Those that mention function match hold only for terms which are well typed in $\emptyset$. In addition, the auxiliary functions are not defined for variables. For this reason, we have proved the lemmas only for terms which are well typed in $\emptyset$.

The following lemma proves that local interactions preserve well-typedness of involved processes and data.

Lemma 3.4.11. *Let* $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$, $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ *and* $(D, R) \rightsquigarrow (D', R')$. *Then*

(1) $\emptyset \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$, *and*

(2) $\emptyset \vdash R' : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

Proof. The proof is by induction on the derivation of $(D, R) \rightsquigarrow (D', R')$.

$\boxed{\text{[L-Communication]}}$ In this case:

- $D' = D$;
- $R = a^{T\mathsf{v}}!\mathsf{v}^{\neg\rho'} \mid a^{T\mathsf{v}}?x.P^{\neg\rho}$;
- $R' = P\{\mathsf{v}/x\}^{\neg\rho}$.

From [T-Role Parallel] and [T-Role] we obtain:

- (P.1) $\emptyset \vdash a^{T\mathsf{v}}!\mathsf{v} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho')$;
- (P.2) $\emptyset \vdash a^{T\mathsf{v}}?x.P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$.

From (P.1) and [T-Output] we obtain (H.1) $\emptyset \vdash \mathsf{v} : T\mathsf{v}$. From (P.2) and [T-Input] we obtain (H.2) $x : T\mathsf{v} \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. From (H.1), (H.2) and Lemma 3.4.1(1) we deduce $\emptyset \vdash P\{\mathsf{v}/x\} : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We conclude (2) by an application of [T-Role], while (1) trivially holds.

$\boxed{\text{[L-Run]}}$ In this case:

- $D' = D$;
- $R = \texttt{RUN}_p{}^{\neg\rho}$;

- $R' = R_1 \mid \ldots \mid R_n$ where $\{R_1, \ldots, R_n\} = \texttt{run}(D, p)$.

From Lemma 3.4.5 we obtain that $\emptyset \vdash R_i : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ for every $i \in \{1, \ldots, n\}$. We conclude (2) by $n - 1$ applications of [T-Role Parallel], while (1) trivially holds.

[L-Read] In this case:

- $D' = D$;
- $R = \texttt{READ}_p(\chi).P^{\neg\rho}$;
- $R' = P\mathrm{s}_1{}^{\neg\rho} \mid \ldots \mid P\mathrm{s}_n{}^{\neg\rho}$ where $\{\mathrm{s}_1, \ldots, \mathrm{s}_n\} = \texttt{read}(D, p, \chi)$.

From [T-Role] and [T-Read] we obtain (H.1) $\Gamma_\chi \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. From Lemma 3.4.6, Lemma 3.4.1(3) and (H.1) we deduce $\emptyset \vdash P\mathrm{s}_i : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ for every $i \in \{1, \ldots, n\}$. Then, we obtain $\emptyset \vdash P\mathrm{s}_i{}^{\neg\rho} : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$ for every $i \in \{1, \ldots, n\}$ by [T-Role]. We conclude (2) with $n - 1$ applications of [T-Role Parallel], while (1) trivially holds.

[L-Change] In this case:

- $R = \texttt{CHANGE}_p(\chi, \mathsf{V}).P^{\neg\rho}$;
- $D' = \texttt{change}(D, p, \chi, \mathsf{V})$;
- $R' = P^{\neg\rho}$.

From [T-Role] we obtain (H.1) $\emptyset \vdash \texttt{CHANGE}_p(\chi, \mathsf{V}).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and then deduce (1) by Lemma 3.4.7. From (H.1) and [T-Change] we get $\emptyset \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We obtain (2) by [T-Role].

[L-Enable] In this case:

- $R = \texttt{ENABLE}_p(r).P^{\neg\rho}$;
- $D' = \texttt{enable}(D, p, r)$;
- $R' = P^{\neg\rho}$.

From [T-Role] we obtain (H.1) $\emptyset \vdash \texttt{ENABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and then deduce (1) by Lemma 3.4.10(1). From (H.1) and [T-Enable] we get $\emptyset \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We obtain (2) by [T-Role].

[L-Disable] In this case:

- $R = \texttt{DISABLE}_p(r).P^{\neg\rho}$;
- $D' = \texttt{disable}(D, p, r)$;
- $R' = P^{\neg\rho}$.

From [T-Role] we obtain (H.1) $\emptyset \vdash \texttt{DISABLE}_p(r).P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$ and then deduce (1) by Lemma 3.4.10(2). From (H.1) and [T-Disable] we get $\emptyset \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho)$. We obtain (2) by [T-Role].

[L-Structural Congruence] In this case we obtain the proof by induction hypothesis and Lemma 3.4.2.

[L-Parallel] In this case we obtain the proof by induction hypothesis and [T-Role Parallel].

[L-Restriction] In this case we obtain the proof by induction hypothesis and [T-Role Restriction]. □

Subject reduction on networks is proved using Lemma 3.4.11.

Theorem 3.4.12 (Subject reduction). *If* $\emptyset \vdash N : \mathit{Network}$ *and* $N \to N'$, *then* $\emptyset \vdash N' : \mathit{Network}$.

Proof. The proof is by induction on the derivation of $N \to N'$.

[R-Local] In this case:

- $N = l[D \parallel R]$
- $(D, R) \rightsquigarrow (D', R')$
- $N' = l[D' \parallel R']$.

From [T-Location] we obtain

- $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$;
- $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$;
- $\emptyset \vdash R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

We derive the proof by Lemma 3.4.11 and [T-Location].

[R-Go] In this case:

- $N = l[D_l \parallel \texttt{GO}\ m.R^{\neg\rho} \mid R_l] \,[]\, m[D_m \parallel R_m]$
- $N' = l[D_l \parallel R_l] \,[]\, m[D_m \parallel R \mid R_m]$

From [T-Network Parallel] we obtain:

- (P.1) $\emptyset \vdash l[D_l \parallel \texttt{GO}\ m.R^{\neg\rho} \mid R_l] : \mathit{Network}$;
- (P.2) $\emptyset \vdash m[D_m \parallel R_m] : \mathit{Network}$.

From (P.1), (P.2) and [T-Location] we obtain:

- $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$;
- $\emptyset \vdash D_l : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$;
- (L.1) $\emptyset \vdash \texttt{GO}\ m.R^{\neg\rho} \mid R_l : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$;

- $\emptyset \vdash m : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$;
- $\emptyset \vdash D_m : \mathit{DataTree}(\sigma', \mathcal{E}', \mathcal{D}', \tau', \zeta')$;
- $\emptyset \vdash R_m : \mathit{RoleProcess}(\sigma', \mathcal{E}', \mathcal{D}')$;

From (L.1), [T-Role Parallel] and [T-Go] we get:

- $\emptyset \vdash R_l : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$;
- $\emptyset \vdash m : \mathit{Location}(\sigma'', \mathcal{E}'', \mathcal{D}'')$;
- $\emptyset \vdash R : \mathit{RoleProcess}(\sigma'', \mathcal{E}'', \mathcal{D}'')$.

[T-Location Name] implies $(\sigma'', \mathcal{E}'', \mathcal{D}'') = (\sigma', \mathcal{E}', \mathcal{D}')$. We derive the proof by applications of typing rules [T-Role Parallel], [T-Location] and [T-Network Parallel].

[R-Stay] In this case:

- $N = l[D \parallel \texttt{GO}\ l.R^{\neg\rho} \mid R']$;
- $N' = l[D \parallel R \mid R']$.

From [T-Location] we obtain

- $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$;
- $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$;
- $\emptyset \vdash \texttt{GO}\ l.R^{\neg\rho} \mid R' : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

Then, from [T-Role Parallel] and [T-Go] we get:

- $\emptyset \vdash R' : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$;
- $\emptyset \vdash l : \mathit{Location}(\sigma', \mathcal{E}', \mathcal{D}')$;
- $\emptyset \vdash R : \mathit{RoleProcess}(\sigma', \mathcal{E}', \mathcal{D}')$.

[T-Location Name] implies $(\sigma', \mathcal{E}', \mathcal{D}') = (\sigma, \mathcal{E}, \mathcal{D})$. We derive the proof by applications of rules [T-Role Parallel] and [T-Location].

[R-Structural Congruence] In this case we obtain the proof by induction hypothesis and Lemma 3.4.2.

[R-Parallel] In this case we obtain the proof by induction hypothesis and [T-Network Parallel].

[R-Network Restriction] In this case we obtain the proof by induction hypothesis and [T-Network Restriction]. □

Corollary 3.4.13. *If $\emptyset \vdash N : \mathit{Network}$ and $N \rightarrow^* N'$, then $\emptyset \vdash N' : \mathit{Network}$.*

We can now prove the soundness of the system.

Theorem 3.4.14 (Type safety). *Let $\emptyset \vdash N : \mathit{Network}$. Then N is well behaved.*

Proof. Let $N \to^* \vec{\nu}(l[D \parallel P^{\neg\rho} \mid R] \,[]\, N')$ and $\mathcal{P}(l) = (\sigma, \mathcal{E}, \mathcal{D})$. From 3.4.13 we know that $\emptyset \vdash \vec{\nu}(l[D \parallel P^{\neg\rho} \mid R] \,[]\, N') : \mathit{Network}$. Then, from [T-Network Restriction] and [T-Network Parallel] we obtain $\emptyset \vdash l[D \parallel P^{\neg\rho} \mid R] : \mathit{Network}$. From [T-Location Name] and [T-Location] we deduce:

- (L.1) $\emptyset \vdash l : \mathit{Location}(\sigma, \mathcal{E}, \mathcal{D})$;
- (L.2) $\emptyset \vdash D : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau, \zeta)$;
- (L.3) $\emptyset \vdash P^{\neg\rho} \mid R : \mathit{RoleProcess}(\sigma, \mathcal{E}, \mathcal{D})$.

From (L.3), [T-Role Parallel] and [T-Role] we get

$$\text{(R.1)}\ \emptyset \vdash P : \mathit{Process}(\sigma, \mathcal{E}, \mathcal{D}, \rho).$$

Now, we prove items of Definition 3.2.25.

(1) Since types of data trees and pure processes are well formed, from (L.2) and (R.1), we know that $\sigma \leqq \tau$ and $\sigma \leq \rho$. By Definition 3.3.3 we know that $\mathcal{C}(D) = \tau$, which concludes the proof.

(2) Straightforward from [T-Enable].

(3) Straightforward from [T-Disable].

(4) We give the proof for $n = 2$. We have only two interesting cases. Let $D = \mathsf{a}_1^{\tau_1}[D']$ and $D' \neq \emptyset_D$. [T-Data Tree] implies $\emptyset \vdash D' : \mathit{DataTree}(\sigma, \mathcal{E}, \mathcal{D}, \tau', \zeta')$ where $\tau_1 \leqq \tau'$. The next edge of the path, $\mathsf{a}_2^{\tau_2}$ must be one of initial edges of D', so $\tau_2 \in \tau'$. By the definition of $\leqq$ we obtain $\tau_1 \leqq \tau_2$. If $D = D_1 \mid D_2$ the whole path must be a path in either D_1 or D_2.

(5) Annotations of edges in a data tree contain role $\top$ and a well-formed policy does not allow this role to be enabled nor disabled.

(6) [T-Output] implies $\emptyset \vdash \mathsf{v} : T\mathsf{v}$ and $\mathcal{C}(T\mathsf{v}) \leq \rho$. By Definition 3.3.3 we know that $\mathcal{C}(\mathsf{v}) = \mathcal{C}(T\mathsf{v})$, which concludes the proof.

(7) From [T-Run], [T-Read], [T-Change], [T-Enable] and [T-Disable], we know that $\emptyset \vdash p : \mathit{Path}(\alpha)$ and $\alpha \leq \rho$. By Definition 3.3.3 we know that $\mathcal{C}(p) = \alpha$, and so, $\mathcal{C}(p) \leq \rho$.

(8) By Definition 3.2.12 we know that path p complies a path from the root of D to W. This implies that the final edge of the path p complies the parent edge of data term W. From $\alpha \leq \rho$ and Lemma 3.2.13 we obtain that the parent edge is accessible to ρ, which by definition makes W accessible to $P^{\neg\rho}$.

(9) Straightforward from [T-Change].

□

3.5 Conclusions and related work

With the aim of application of RBAC to peer-to-peer model of semi-structured web data, in [40], we introduced the ®X$d\pi$ calculus, a formal model for dynamic web applications with RBAC, and proposed a type system to control its safety. In our framework, a network is a parallel composition of locations, where each location contains processes with roles and a data tree whose edges are associated with roles. Processes can communicate, migrate from a location to another, use the data, change the data and the roles in the local tree. By assigning roles both to data trees and processes, we obtain a model that allows role administration (i.e. activation and deactivation). The language shown in this chapter is essentially the one from [40], except that the calculus presented here, besides the use of slightly different notation, allows replication of any process and gives an alternative definition of reduction relation using local interaction relation, instead of reduction contexts. The proposed type system ensures that the specified location policies and data consistency are preserved during computations and that processes do not exceed rights of their roles in sending, accessing and changing data.

The syntax of processes and operational semantics are inspired by XPath [9], XQuery [16, 126] and XQuery Update Facility [125]. The data model is an edge-labelled rooted tree, similar to [10]. The calculus presented in this chapter and in [40] extend X$d\pi$-calculus [55] with RBAC. The X$d\pi$ calculus models process communication, and process migration, as distributed π-calculus [72], and local interaction between processes and data. A network is a parallel composition of locations, where each location contains one process and one data tree. The paths of X$d\pi$-calculus have richer syntax, there is only one process for data management and semantics is given by a reduction relation closed under structural congruence and reduction contexts. The language presented in [41] extends X$d\pi$-calculus with security levels. KLAIM [37] is a language designed to program distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces or databases.

RBAC has been first formalized in [47]. The current standard is defined by the InterNational Committee for Information Technology Standards in [6]. There is a large literature on models, extensions and implementations for RBAC, we only mention [129, 46, 113, 48, 96]. We discussed a model in which the pure processes are the users, the permissions are the accesses to data in trees and the administration policies of locations prescribe how the association between roles and data can change. Note that we do not have user identifiers, and so we cannot activate and deactivate roles for users. Our design choice is motivated by the focus on the interaction between processes and data trees and to the best of our knowledge this is a first attempt in this direction. Other common features of RBAC system we did not consider here, since we could smoothly add them to the present calculus, are: incompatible roles, static and dynamic separation of roles, limits on the number of users authorised for a given role.

Security policies are often concerned with information flow that cannot be implemented correctly with classical access control methods which essentially prevent unauthorized access. In particular, an overall behavior check of the program must

be performed, including the part that has not executed jet, in order to enforce policies which depend on how obtained data will be used in the future execution of the program. A language extending KLAIM which can be used for enforcing such predictive access policies is proposed in [153]. The type system presented in this chapter checks all the actions of the processes and it is used to control the usage of the data.

Access control has been studied in various forms for many calculi modelling concurrent and distributed systems. Sophisticated types controlling the use of resources and the mobility of processes have been proposed for the $D\pi$ calculus [72, 70]. In $D\pi$ the resources are channels which support binary communication between processes. The typing system guarantees that distributed processes cannot access the resources without first being granted the capability to do so. Processes can augment their sets of capabilities via communication with other processes. In the SafeDpi calculus [71] parametrised code may be sent between locations and types restrict the capabilities and access rights of any processes launched by incoming code. The paper [41] discusses a type system for the X$d\pi$ calculus based on security levels: security levels in data trees are assigned only to the data in the leaves. Co-actions have been introduced for ambient calculi as a basic mechanism for regulating access to locations and use of their resources [98, 29, 56]. More refined controls for ambient calculi include passwords [103, 21], classifications in groups [27, 35], mandatory access control policies [20], membranes regulating the interaction between computing bodies and external environments [63]. Access control for binary sessions without delegation has been investigated in [95]. Information flow safety in multiparty sessions has been studied in [22]. In that paper a calculus for multiparty sessions is enriched with security levels for messages and a monitored semantics for this calculus, which blocks the execution of processes as soon as they attempt to leak information is proposed.

Closely related papers are [18], [33] and [41]. The authors of [18] equip the π-calculus with the notion of user: they tag processes with names of users and with sets of roles. Processes can activate and deactivate roles. A mapping between roles and users and a mapping between read/write actions and roles control access rights. A type discipline statically guaranties that systems not respecting the above mappings are rejected. In [33], the authors define a boxed ambient calculus extended with a distributed RBAC mechanism where each ambient controls its own access policy. A process is associated with an owner and a set of activated roles that grant permissions for mobility and communication. The calculus includes primitives to activate and deactivate roles. The behavior of these primitives is determined by the process's owner, its current location and its currently activated roles. In [41] the authors add security levels to X$d\pi$ calculus. The type system ensures that the data in a location are accessible only to processes in locations of equal or higher security level. Processes originating in a location can only go to locations of equal or less security level, with the exception of movements which are returns to the source location.

Chapter 4

Types for private access

Overview This chapter is based on the paper "Linked Data Privacy" [90] which introduces a formal model of linked data that can statically detect run-time errors due to privacy violations. Here, the terminology and the language are slightly adapted, and the proofs are given in more details. In Section 4.1 we describe the setting and the problem we are addressing. In Section 4.2 we introduce a new calculus for modelling the web of linked data and we define well-behaved network. In Section 4.3 we introduce a simple, yet sufficiently powerful type assignment system which, together with the introduced policy order, is able to statically check whether a network is well behaved. In Section 4.4 first we prove several auxiliary properties and then the main result - that a well-typed network is well behaved. Section 4.5 concludes the chapter with discussion and related work.

4.1 Linked data and private access

The more data is connected with other sources of information, the more its value increases. Having that in mind, an initiative to establish a generic format for connecting data, called the Semantic Web, was born one decade ago. It has grown into a collection of recommendations for publishing data on the Web [15, 14]. The Web of Linked Data is expected to expand into a huge graph of linked data, based on four principles defined by Berners-Lee [11]:

1. the use of URIs (IRIs) to name things,
2. the use of HTTP URIs (IRIs) so that these names may be found,
3. the use of standards in order to provide functional information about things,
4. the possibility of making connections between the data in order to create rich web in which one can find all kinds of things.

The recommended standards for publishing and querying linked data are *Resource Description Framework (RDF)* [101, 131] and *SPARQL* [122].

RDF is a framework for representing information about *resources*. Anything can be considered as a resource, including documents, people, physical objects and

abstract concepts. Publishing data in an open standard such as RDF and interlinking data sources aims to transform Web of Documents into more (re)usable, machine-readable, Web of Data. The essence of RDF data model [36] are *triples*

$$< \texttt{subject} > \ < \texttt{predicate} > \ < \texttt{object} >$$

which allow one to make statements about resources. An RDF triple expresses a relationship between the subject and the object. The nature of their relationship is specified by the predicate and is called a property. RDF data are sets of triples which may be understood as directed graph such that subjects and objects are nodes and predicates are the edge labels. The edges are directed from subject to object. The elements of triples are allowed be *URIs (IRIs)*, *literals* or *blank nodes*. The abbreviation IRI stands for International Resource Identifier, which is a string of Unicode characters specified by a standard. The abbreviation URI stands for Uniform Resource Identifier which is, a less general, string of ASCII characters. Literals are basic values that are not IRIs. Blank nodes are used to represent something that exists but whose name is unknown or insignificant. The subject and the object of a triple may be IRIs, literals or blank nodes, while the predicate may only be an IRI.

There are several concrete syntaxes for writing down RDF graphs (Turtle [8], JSON-LD [133], RDFa [73], RDF/XML [54] etc.) They all meet the criteria of [36] and are convenient for working with RDF documents, since they may encode the same RDF data in many different ways.

The recommended query language for RDF is SPARQL. Its query forms enable querying *graph patterns* along with their conjunctions and disjunctions. Features such as negation, subqueries, creation of values by expressions, extensible value test and constraint of queries by source RDF graph are also supported. The basic graph pattern is the *triple pattern*. The syntax of the triple patterns is essentially the same as the syntax of the RDF triples, except that variables may appear as any of the triples' elements. SPARQL Update [59] is an extension of SPARQL that provides operations to insert, delete and update RDF data.

A great merit of the Web of Linked Data is its exposure to public consumption. Even though public availability brings a great advantage to users of such data, not all data are produced for public usage. For example, RDF is often used to represent personal information and data from social networks. This gives rise to the question of privacy of linked data, since the lack of privacy protection mechanisms often discourages people from publishing data on the Web of Linked Data. Addressing this issue requires a clear explanation for the intuition of the notion of *privacy*. In [151], the privacy is defined as "the ability to control who has access to information and to whom that information is communicated". In this sense, we deem that privacy of data is protected in case:

- an owner of the data can always access his own data, and
- an owner of the data can control access to his data, i.e. he can create conditions for other consumers to access parts or all of his data, and
- an owner of the data can change the data.

```
< Alice > < is > < person >
< Alice > < has affiliation > < UNS >
< UNS > < organizes > < event >
< Bob > < organizes > < event >
< paper > < published > < journal >
```

Figure 4.1: RDF triples in pseudocode

Furthermore, privacy may not include private status of some data only, but also, whether the data are significant or not for some group and whether the readers are able to understand the data properly [134].

There are several works on formalizations of different aspects of linked data. The calculi that appear in [81, 82] provide an abstract syntax for RDF and SPARQL in order to capture Linked Data structures and queries. The authors of [127, 128] describe the formal semantic model of Privacy Preference Ontology and present a privacy preference manager that lets users create privacy preferences by means of the aforementioned ontology and restrict access to their data to third-party users based on profile data features. In [42] the authors study provenance for Linked Data in a process calculus, where types statically evaluate provenance driven access control. In [90] we focus on privacy of linked data in terms of access control. We give the brief summary of the research presented in [90] and in this chapter:

- we formalize a calculus for modelling a linked data network;
- we develop a type system for preventing privacy violations and we show its soundness.

4.1.1 Examples and motivation

In Figure 4.1, we give an example in the abstract syntax of RDF data model [36, 131] which is the most convenient for the formal reasoning. The name `UNS` appears as an object in one triple and a subject in another. Examples of SPARQL queries executed on this data are shown in Figure 4.2.

In general, the results of queries can be boolean values, sets or RDF graphs. The first ask query after the key word `ASK` has, inside the curly braces, the simplest triple pattern - a single triple that has no variables. This query should be understood as the question: is there a triple `< Bob > < is > < person >` in the queried data? The result is boolean value `FALSE`. The second ask query, inside the curly braces, has a pattern consisting of two triples connected with the key word `UNION`. This query should be understood as the question: is there a triple `< Alice > < is > < person >` or a triple `< Bob > < is > < person >` in the queried data? In this case, the result is `TRUE`.

The select query in Figure 4.2 has two parameters: the variable `?x` after the key word `SELECT`, and the pattern after the key word `WHERE`. The pattern is such that the subject is the variable, the predicate is `organizes` and the object is

Query	Result
`ASK { < Bob > < is > < person > }`	`FALSE`
`ASK { < Alice > < is > < person > UNION < Bob > < is > < person > }`	`TRUE`
`SELECT ?x WHERE { <?x > organizes event }`	?x UNS Bob

Figure 4.2: ASK and SELECT queries

`event`. The triples in the queried data, identified by this pattern, are $<$ `UNS` $>$ $<$ `organizes` $>$ $<$ `event` $>$ and $<$ `Bob` $>$ $<$ `organizes` $>$ $<$ `event` $>$. The query results in the set of substitutions for the variable ?`x`, such that the first element of each identified triple substitutes the variable.

Figure 4.3 contains examples of two SPARQL Update queries. Both queries are executed on the data from Figure 4.1. The delete query has the pattern after the key word `WHERE` that identifies the triples in the queried data that should be deleted. The dot in the pattern denotes the conjunction of two patterns. This is an example of a query that removes all the triples about anyone who is a person. The insert data operation adds triples to the queried data.

Now, let us consider a network of users, each having their own profile (data) in RDF format and processes running on their behalfs. In order to enable each owner of data to control privacy of its data, we assign a *privacy protection policy* to each user name, resource and data triple. Similarly to [127, 128], we take SPARQL patterns as privacy protection policies and say that a user can access a data triple if the user's data satisfy the `ASK` query of the policy assigned to the triple. If a user name is accessible to different users, they are all considered as the data owners and the data can be changed by any of them. A privacy violation would be changing the data in such way that they become inaccessible to its original owner. For example, both queries from Figure 4.3 are potentiality dangerous. The delete query might delete some crucial data, or it might not be authorized to change the data at all. The insert data query might add a data triple with a policy such that owner's data do not satisfy the ASK query of that policy. In addition, privacy violations may be caused by uncontrolled reading of the data.

In the following, we give the formalization of this scenario and we develop a type system for preventing the aforementioned privacy violations.

Query

```
DELETE
WHERE
{
?x  < is >  < person >.
?x  ?y  ?z
}
```

Data after

```
< UNS >  < organizes >  < event >
< Bob >  < organizes >  < event >
< paper >  < published >  < journal >
```

Query

```
INSERT DATA
{
< photo >  < of >  < Alice >
}
```

Data after

```
< Alice >  < is >  < person >
< Alice >  < has affiliation >  < UNS >
< UNS >  < organizes >  < event >
< Bob >  < organizes >  < event >
< paper >  < published >  < journal >
< photo >  < of >  < Alice >
```

Figure 4.3: DELETE and INSERT DATA queries

U	::=		**Policy**
		(u, u, u)	triple
	|	$U \vee U$	disjunction
	|	$\exists x.U$	exists

Figure 4.4: Syntax of privacy protection policies

D	::=		**Data**	χ	::=		**Pattern**
		$\emptyset_D$	empty			$(u, u, u)^U$	triple
	|	$(a, a, a)^U$	triple		|	$\chi \vee \chi$	disjunction
	|	$D \mid D$	parallel		|	$\exists x^U.\chi$	exists

Figure 4.5: Syntax of data and data patterns

4.2 Language

Notation Let us assume that there is an infinite set IRIs of *IRI names (IRIs)* ranged over by $a, b, c, \ldots$, an infinite set $\mathsf{Variables}$ of *variables* ranged over by $x, y, \ldots$ We let symbols $u, v, \ldots$ range over elements of $\mathsf{IRIs} \cup \mathsf{Variables}$, and refer to them as *names*.

4.2.1 Syntax

Policies We assume given a function $\mathcal{P}(\cdot)$ that assigns *privacy protection policies* to IRI names. The syntax of privacy protection policies is given by the grammar in Figure 4.4. The term (u_1, u_2, u_3) denotes a triple of IRI names or variables, the term $U_1 \vee U_2$ denotes disjunction between policies U_1 and U_2, and the term $\exists x.U$ denotes an exists policy in which variable x is bounded. We say that *a privacy protection policy is well formed* if it contains no occurrences of free variables and we consider only such policies. We use $U, V, W, \ldots$ to range over privacy protection policies. Sometimes we refer to a privacy protection policy as privacy policy or just policy.

Data The syntax of *data* is given in Figure 4.5. The term $\emptyset_D$ denotes empty data, the term $(a_1, a_2, a_3)^U$ denotes a triple of IRI names with an associated privacy protection policy U and the term $D_1 \mid D_2$ denotes a parallel composition of data. We use $D, E, \ldots$ to range over data. Data variables are ranged over by $X, Y, \ldots$ We let symbol δ range over data and data variables.

Patterns The syntax of *data patterns*, given in Figure 4.5, is very similar to the syntax of privacy protection policies. The only difference is that patterns are decorated by policies. We use $\chi, \psi, \omega, \ldots$ to range over data patterns. We often refer to data patterns as just patterns.

P	::=		**Process**	N	::=		**Network**
		0	inaction			$a[D \parallel P]$	user
	$\mid$	$P \mid P$	parallel		$\mid$	$N \,[]\, N$	parallel
	$\mid$	$P \oplus P$	choice				
	$\mid$	$*P$	replication				
	$\mid$	$\texttt{READ}_u(\chi, X).P$	read data				
	$\mid$	$\texttt{WRITE}_u(\delta).P$	write data				
	$\mid$	$\texttt{CLEAR}_u$	delete data				
	$\mid$	$\texttt{MODIFY}_u(\chi, D).P$	change data				
	$\mid$	$\texttt{SELECT}_u(\exists x^U.\chi, x).P$	select name				
	$\mid$	$\texttt{UPDATE}_u(\chi, U).P$	update policy				

Figure 4.6: Syntax of processes and networks.

Processes The syntax of *processes* is given in Figure 4.6. We can divide them into three groups:

1. the terms 0, $P_1 \mid P_2$, $P_1 \oplus P_2$ and $*P$ denote π-calculus processes;

2. the terms $\texttt{READ}_u(\chi, X).P$, $\texttt{WRITE}_u(\delta).P$, $\texttt{CLEAR}_u$, $\texttt{MODIFY}_u(\chi, D).P$ and $\texttt{SELECT}_u(\exists x^U.\chi, x).P$ denote processes that manage the data.

   In particular, the term $\texttt{READ}_u(\chi, X).P$ denotes a process that reads from the user named u the data identified by the pattern χ. The data variable X is substituted with the identified data in the continuation P.

   The term $\texttt{WRITE}_u(\delta).P$ denotes a process that adds the data δ to the data of user named u and then continues as P.

   The term $\texttt{CLEAR}_u$ denotes a process that deletes the entire data of the user named u.

   The term $\texttt{MODIFY}_u(\chi, D).P$ denotes a process that deletes from the user named u the data identified by the pattern χ, and writes D to the same user and then continues as P.

   The term $\texttt{SELECT}_u(\exists x^U.\chi, x).P$ denotes a process that selects specific elements of data triples identified by the pattern χ. The variable x is substituted with the selected names in the continuation P. Any other exists pattern that may appear inside the pattern χ is not considered as a binder for P.

3. the term $\texttt{UPDATE}_u(\chi, U).P$ denotes a process that manages privacy protection policies. In particular, it replaces by U the privacy policies on the part of user's u data, identified by the pattern χ and then continues as P.

We use $P, Q, \ldots$ to range over processes.

Networks The syntax of *networks* is given in Figure 4.6. The term $a[D \parallel P]$ denotes a *user* named a that encloses data D and process P. The term $N_1 \,[]\, N_2$ denotes a parallel composition of networks. We say that *a network is well formed* if all its users have different names.

User `Alice`		$\mathcal{P}(\texttt{Alice}) = (\texttt{Alice}, \texttt{is}, \texttt{person}) = U_{\texttt{Alice}}$
$D_{\texttt{Alice}}$	$=$	$(\texttt{Alice}, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}}$
	$\mid$	$(\texttt{Alice}, \texttt{has affiliation}, \texttt{UNS})^{U_{\texttt{Alice}}}$
	$\mid$	$(\texttt{UNS}, \texttt{organizes}, \texttt{event})^{U_{\texttt{P}}}$
	$\mid$	$(\texttt{Bob}, \texttt{organizes}, \texttt{event})^{U_{\texttt{P}}}$
	$\mid$	$(\texttt{paper}, \texttt{published}, \texttt{journal})^{U_{\texttt{Alice}} \vee \exists x.(x, \texttt{is}, \texttt{researcher})}$
User `Bob`		$\mathcal{P}(\texttt{Bob}) = (\texttt{Bob}, \texttt{is}, \texttt{person}) = U_{\texttt{Bob}}$
$D_{\texttt{Bob}}$	$=$	$(\texttt{Bob}, \texttt{has affiliation}, \texttt{UNS})^{U_{\texttt{P}}}$
	$\mid$	$(\texttt{Bob}, \texttt{is}, \texttt{researcher})^{U_{\texttt{P}}}$
	$\mid$	$(\texttt{Bob}, \texttt{is a friend of}, \texttt{Cindy})^{U_{\texttt{P}}}$

Figure 4.7: Running example of data in a network

Syntactic conventions We adopt some standard conventions regarding the syntax of processes:

- we sometimes use a prefix form for parallel compositions and write, for example, $\prod_{i=1..n} P_i$ instead of $P_1 \mid \cdots \mid P_n$;
- we identify $\prod_{i \in \emptyset} P_i$ with 0;
- we omit trailing occurrences of 0 and write, for example, $\texttt{WRITE}_a(D)$ instead of $\texttt{WRITE}_a(D).0$.

Example 4.2.1 (Running example). In this and in several following examples, we consider the following network:

$$\texttt{Alice}[D_{\texttt{Alice}} \parallel P_{\texttt{Alice}}] \quad [] \quad \texttt{Bob}[D_{\texttt{Bob}} \parallel P_{\texttt{Bob}}]$$

Figure 4.7 shows the data running on behalf of both users in the network, `Alice` and `Bob`. The triples of user `Alice` are similar to those given in Figure 4.1. The difference is that, in our setting, the triples are associated with privacy protection policies. Let us assume that the privacy protection policy of IRI names `Alice` and `Bob` is as follows: $\mathcal{P}(\texttt{Alice}) = (\texttt{Alice}, \texttt{is}, \texttt{person})$ and $\mathcal{P}(\texttt{Bob}) = (\texttt{Bob}, \texttt{is}, \texttt{person})$. Further, let us assume that all other IRIs in this example have the same privacy protection policy: $U_{\texttt{P}} = \exists x.\exists y.\exists z.(x, y, z)$. We leave the specification of processes $P_{\texttt{Alice}}$ and $P_{\texttt{Bob}}$ for the further examples. △

Example 4.2.2. The processes we consider in the calculus are closely related to SPARQL and SPARQL Update queries. We give one possible encoding of select, delete and insert from Figures 4.2 and 4.3 without specifying policies.
$\texttt{SELECT}_{\texttt{Alice}}(\exists x^{U}.(x, \texttt{organizes}, \texttt{event})^{V}, x)$
$\texttt{SELECT}_{\texttt{Alice}}(\exists x^{U_1}.(x, \texttt{is}, \texttt{person})^{V}, x).\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_2}.\exists z^{U_3}.(x, y, z)^{W}, \emptyset_D)$
$\texttt{WRITE}_{\texttt{Alice}}((\texttt{photo}, \texttt{at}, \texttt{party})^{U})$
We will use these processes to illustrate the semantics of our language. We will, also, show that they are the examples of processes that may violate privacy in a network. △

4.2.2 Satisfaction of ask queries

In Section 4.1 we informally introduced two notions: satisfaction of a privacy protection policy and identification of data triples. Now, we give their precise definitions. The first one is a deductive system for checking whether some data *satisfies a privacy protection policy.*

Definition 4.2.3 (Satisfaction of a privacy policy). *We say that data D satisfies a privacy policy U, written $D \models \texttt{ASK}(U)$, if it can be deduced by means of the following rules:*

$$\text{[A-Triple]}\quad (a,b,c)^W \models \texttt{ASK}((a,b,c))$$

$$\text{[A-OrL]}\quad \frac{(a,b,c)^W \models \texttt{ASK}(U)}{(a,b,c)^W \models \texttt{ASK}(U \vee V)} \qquad \text{[A-OrR]}\quad \frac{(a,b,c)^W \models \texttt{ASK}(V)}{(a,b,c)^W \models \texttt{ASK}(U \vee V)}$$

$$\text{[A-Exists]}\quad \frac{(a,b,c)^W \models \texttt{ASK}(U\{d/x\})}{(a,b,c)^W \models \texttt{ASK}(\exists x.U)} \qquad \text{[A-Ask]}\quad \frac{D \equiv (a,b,c)^W \mid D' \qquad (a,b,c)^W \models \texttt{ASK}(U)}{D \models \texttt{ASK}(U)}$$

Rule [A-Triple] states that data triple $(a,b,c)^W$ satisfies privacy policy (a,b,c).

Rules [A-OrL] and [A-OrR] state that a data triple satisfies disjunction of two policies if is satisfies one of them.

Rule [A-Exists] state that a data triple satisfies exists policy $\exists x.U$ if there exists IRI name d such that the data triple satisfies policy $U\{d/x\}$. For example, $(a,b,c)^W \models \texttt{ASK}(\exists x.(x,b,c))$ because $(x,b,c)\{a/x\} = (a,b,c)$ and by [A-Triple] we know $(a,b,c)^W \models \texttt{ASK}((a,b,c))$.

Rule [A-Ask] states that the data satisfy a privacy protection policy if there is at least one triple in the data that satisfies the policy. We formalize this in the following lemma which is proven straightforwardly from [A-Ask].

Lemma 4.2.4. *If $D \models \texttt{ASK}(U)$, then there exists a triple $(a,b,c)^W$ such that $D \equiv (a,b,c)^W \mid D'$ and $(a,b,c)^W \models \texttt{ASK}(U)$.*

We use notion of policy satisfaction to check whether some user is allowed to access a data triple. More precisely, we say that a user $a[D \parallel P]$ is allowed to access a data triple with privacy protection policy U if $D \models \texttt{ASK}(U)$.

As a direct consequence of Lemma 4.2.4, we have that a user with empty data is not allowed to access any data triple.

Lemma 4.2.5. *If $D \models \texttt{ASK}(U)$, then $D \neq \emptyset_D$.*

We call users whose corresponding data is empty *blocked users.* All other users are called *active users.*

Example 4.2.6. The ask queries for Figure 4.2 are encoded into our calculus as:
$\texttt{ASK}((\texttt{Bob}, \texttt{is}, \texttt{person}))$
$\texttt{ASK}((\texttt{Alice}, \texttt{is}, \texttt{person}) \vee (\texttt{Bob}, \texttt{is}, \texttt{person}))$.
It is easy to check that

$$D_{\texttt{Alice}} \models \texttt{ASK}((\texttt{Bob}, \texttt{is}, \texttt{person}))$$

does not hold, while

$$D_{\texttt{Alice}} \models \texttt{ASK}((\texttt{Alice}, \texttt{is}, \texttt{person}) \vee (\texttt{Bob}, \texttt{is}, \texttt{person}))$$

holds. Furthermore, it also easy to check that

$$D_{\texttt{Alice}} \models \texttt{ASK}(\exists x.(x, \texttt{is}, \texttt{person}))$$

holds. △

Example 4.2.7 (Public data). We say that a name or a data triple is public if it has privacy protection policy

$$\exists x.\exists y.\exists z.(x, y, z)$$

since for any $D \neq \emptyset_D$ it holds

$$D \models \texttt{ASK}(\exists x.\exists y.\exists z.(x, y, z)).$$

△

Patterns appear as arguments in processes, where they are used to identify the data. The second notion, we now formalize, is a method for checking whether some data triple *satisfies a pattern.*

Definition 4.2.8 (Satisfaction of a pattern). *We say that a triple* $(a, b, c)^W$ *satisfies pattern* χ*, written* $(a, b, c)^W \models \chi$*, if it can be deduced by means of the following rules:*

$$\text{[P-Triple]}\quad (a, b, c)^W \models (a, b, c)^W \qquad \text{[P-OrL]}\quad \frac{(a, b, c)^W \models \chi}{(a, b, c)^W \models \chi \vee \psi} \qquad \text{[P-OrR]}\quad \frac{(a, b, c)^W \models \psi}{(a, b, c)^W \models \chi \vee \psi}$$

$$\text{[P-Exists]}\quad \frac{(a, b, c)^W \models \chi\{d/x\} \quad \mathcal{P}(d) = U}{(a, b, c)^W \models \exists x^U.\chi}.$$

Rules are very similar to those in Definition 4.2.3. Rule [P-Triple] states that in order for a data triple to satisfy a triple pattern, the data triple must be exactly the same as the pattern, including the associated policies. Rule [P-Exists] requires that name d which provides pattern satisfaction has the appropriate privacy policy.

Both notions of satisfaction use essentially the same mechanism, motivated by ASK query of SPARQL. Pattern satisfaction is more strict because the syntax of patterns, besides the structure of the data, includes the privacy policies of triples and names and because it is defined only on single data triples.

Example 4.2.9. The satisfaction of patterns depends on the specified policies of names and triples. The pattern $\exists x^{U_\text{P}}.(x, \texttt{organizes}, \texttt{event})^{U_\text{P}}$ is satisfied only by one data triple of user `Alice`. Namely, the triple $(\texttt{UNS}, \texttt{organizes}, \texttt{event})$. △

To ease the formalization, our language supports only a set of features important for privacy protection of data in terms of access control. In particular,

$$D \mid \emptyset_D \equiv D \qquad D_1 \mid D_2 \equiv D_2 \mid D_1 \qquad (D_1 \mid D_2) \mid D_3 \equiv D_1 \mid (D_2 \mid D_3)$$

$$P \mid 0 \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$*P \equiv P \mid *P \qquad N_1 \,[]\, N_2 \equiv N_2 \,[]\, N_1 \qquad (N_1 \,[]\, N_2) \,[]\, N_3 \equiv N_1 \,[]\, (N_2 \,[]\, N_3)$$

Figure 4.8: Structural congruence

we consider only triple, disjunction and exists policies and patterns while in the syntax of SPARQL patterns there are several more. The syntax can be extended with, for example *conjunction*, and both notions of satisfaction can be redefined accordingly. We would still be able to elegantly check whether the data satisfy a policy, while identifying the exact data that matches the pattern would get more complicated. Since the conjunction pattern does not bring anything essentially important for the problems considered in this chapter, it is omitted. For a similar reason, the syntax of data allows only data triples, while graphs are excluded. The processes are related to some of SPARQL and SPARQL Update queries, as we have seen in the examples.

4.2.3 Operational semantics

The operational semantics is defined in terms of a structural congruence over data, processes and networks, an interaction relation and a reduction relation. Structural congruence identifies structurally equivalent terms. It is the smallest relation $\equiv$ including alpha conversion (renaming of bound variables) and the laws in Figure 4.8, stating that parallel composition of data, processes and networks is commutative, associative, and has $\emptyset_D$ and 0 as neutral element for data and processes, respectively, and that a replicated process may be unfolded. The binders of the calculus are commands for reading and selecting data, which bind name variables and data variables, respectively.

The operational semantics for data and data policies management depends on the auxiliary functions given in Figure 4.9. These functions identify the part of the data that satisfies a pattern, using satisfaction relations, and do changes if required. The function

- `readable` takes data as both arguments and returns the parallel composition of triples of the second data that a process running on behalf of the first data can access. We will say that an user with data D_a can *fully access* the data D_b if $\texttt{readable}(D_a, D_b) = D_b$;
- `read` takes a pattern and data as arguments and returns the parallel composition of the data triples that satisfy the pattern;
- `delete` takes a pattern and data as arguments and, contrary to `read`, returns the parallel composition of all triples that do not satisfy the pattern. In other words, this function deletes the triples that do satisfy the pattern;

$$
\begin{array}{lcl}
\texttt{readable}(D, \emptyset_D) & = & \emptyset_D \\
\texttt{readable}(D, (a,b,c)^V) & = & \begin{cases} (a,b,c)^V & \text{if } D \models \texttt{ASK}(V), \\ \emptyset_D & \text{otherwise} \end{cases} \\
\texttt{readable}(D, D_1 \mid D_2) & = & \texttt{readable}(D, D_1) \mid \texttt{readable}(D, D_2)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{read}(\chi, \emptyset_D) & = & \emptyset_D \\
\texttt{read}(\chi, (a,b,c)^V) & = & \begin{cases} (a,b,c)^V & \text{if } (a,b,c)^V \models \chi, \\ \emptyset_D & \text{otherwise} \end{cases} \\
\texttt{read}(\chi, D_1 \mid D_2) & = & \texttt{read}(\chi, D_1) \mid \texttt{read}(\chi, D_2)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{delete}(\chi, \emptyset_D) & = & \emptyset_D \\
\texttt{delete}(\chi, (a,b,c)^V) & = & \begin{cases} \emptyset_D & \text{if } (a,b,c)^V \models \chi, \\ (a,b,c)^V & \text{otherwise} \end{cases} \\
\texttt{delete}(\chi, D_1 \mid D_2) & = & \texttt{delete}(\chi, D_1) \mid \texttt{delete}(\chi, D_2)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{select}(\exists x^U.\chi, \emptyset_D) & = & \emptyset \\
\texttt{select}(\exists x^U.\chi, (a,b,c)^V) & = & \{\{d/x\} \mid (a,b,c)^V \models \chi\{d/x\} \\
 & & \qquad \wedge\ d \in \{a,b,c\}\ \wedge\ \mathcal{P}(d) = U\} \\
\texttt{select}(\exists x^U.\chi, D_1 \mid D_2) & = & \texttt{select}(\exists x^U.\chi, D_1) \cup \texttt{select}(\exists x^U.\chi, D_2)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{update}(\chi, \emptyset_D, W) & = & \emptyset_D \\
\texttt{update}(\chi, (a,b,c)^V, W) & = & \begin{cases} (a,b,c)^W & \text{if } (a,b,c)^V \models \chi, \\ (a,b,c)^V & \text{otherwise} \end{cases} \\
\texttt{update}(\chi, D_1 \mid D_2, W) & = & \texttt{update}(\chi, D_1, W) \mid \texttt{update}(\chi, D_2, W)
\end{array}
$$

Figure 4.9: Definitions of auxiliary functions.

- `select` takes an existential pattern and data as arguments and returns a set of substitutions. For each triple in the data that satisfies the pattern, a substitution is added to the set, if the selected name has the proper privacy policy. Sometimes we denote a substitution with s and a set of substitutions with S;

- `update` takes a pattern, data and a policy as arguments and returns the data with privacy protection policies updated on triples that satisfy the pattern. The new privacy protection policy is the third argument.

Example 4.2.10. We recall the network from Example 4.2.1. User `Alice` can fully access her own data because it holds $\texttt{readable}(D_{\texttt{Alice}}, D_{\texttt{Alice}}) = D_{\texttt{Alice}}$. User `Bob` is allowed to access some data triples of user `Alice`. In particular

$$\begin{aligned}\texttt{readable}(D_{\texttt{Bob}}, D_{\texttt{Alice}}) \;=\; & (\texttt{UNS}, \texttt{organizes}, \texttt{event})^{U_{\mathsf{P}}} \\ & |(\texttt{Bob}, \texttt{organizes}, \texttt{event})^{U_{\mathsf{P}}} \\ & |(\texttt{paper}, \texttt{published}, \texttt{journal})^{U_{\texttt{Alice}} \vee \exists x.(x, \texttt{is}, \texttt{researcher})}\end{aligned}$$

We give three more examples of auxiliary functions results:

- $\texttt{select}(\exists x^{U_{\texttt{Alice}}}.(x, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}}, D_{\texttt{Alice}}) \;=\; \{\texttt{Alice}/x\}$

- $\texttt{delete}(\exists y^{U_{\mathsf{P}}}.\exists z^{U_{\mathsf{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, D_{\texttt{Alice}})$
  $= (\texttt{UNS}, \texttt{organizes}, \texttt{event})^{U_{\mathsf{P}}}$
  $|(\texttt{Bob}, \texttt{organizes}, \texttt{event})^{U_{\mathsf{P}}}$
  $|(\texttt{paper}, \texttt{published}, \texttt{journal})^{U_{\texttt{Alice}} \vee \exists x.(x, \texttt{is}, \texttt{researcher})}$

- $\texttt{delete}(\exists y^{U_{\mathsf{P}}}.\exists z^{U_{\mathsf{P}}}.(\texttt{Alice}, y, z)^{U_{\mathsf{P}}}, D_{\texttt{Alice}}) = D_{\texttt{Alice}}$

△

We define an interaction relation $\rightsquigarrow_{a,b}$ for each two names $a, b \in \mathsf{IRIs}$ in order to describe interactions between the process running on the behalf of the user $a[D_a \parallel P_a]$ and data of the user $b[D_b \parallel P_b]$. Users with names a and b are not necessarily different. If they are the same, the interaction relation $\rightsquigarrow_{a,a}$ describes the interaction of a user with its own data.

The interaction relation is given by the rules in Figure 4.10. Rules [I-Parallel], [I-Choice] and [I-Structural] are essentially the same as [R-Parallel], [R-Choice] and [R-Structural] reduction rules for π-calculus, given in Figure 2.5, except that they refer to both processes and data. Parallel composition of processes may change the data.

In rule [I-Read], the data D are readable for the user with data D_a and all their triples satisfy the pattern χ. The data D substitute X in the continuation process P.

In rule [I-Write], the data D are composed in parallel with D_b. The premise $D_b \neq \emptyset_D$ disables writing additional data to a blocked user.

In rule [I-Clear], the data D_b is replaced with $\emptyset_D$, meaning that the data of user b is entirely erased, i.e. user b is blocked. Such user is not allowed to access any data, since from the definition of function $\texttt{readable}(\cdot, \cdot)$ and Lemma 4.2.5 we derive $\texttt{readable}(\emptyset_D, D) = \emptyset_D$.

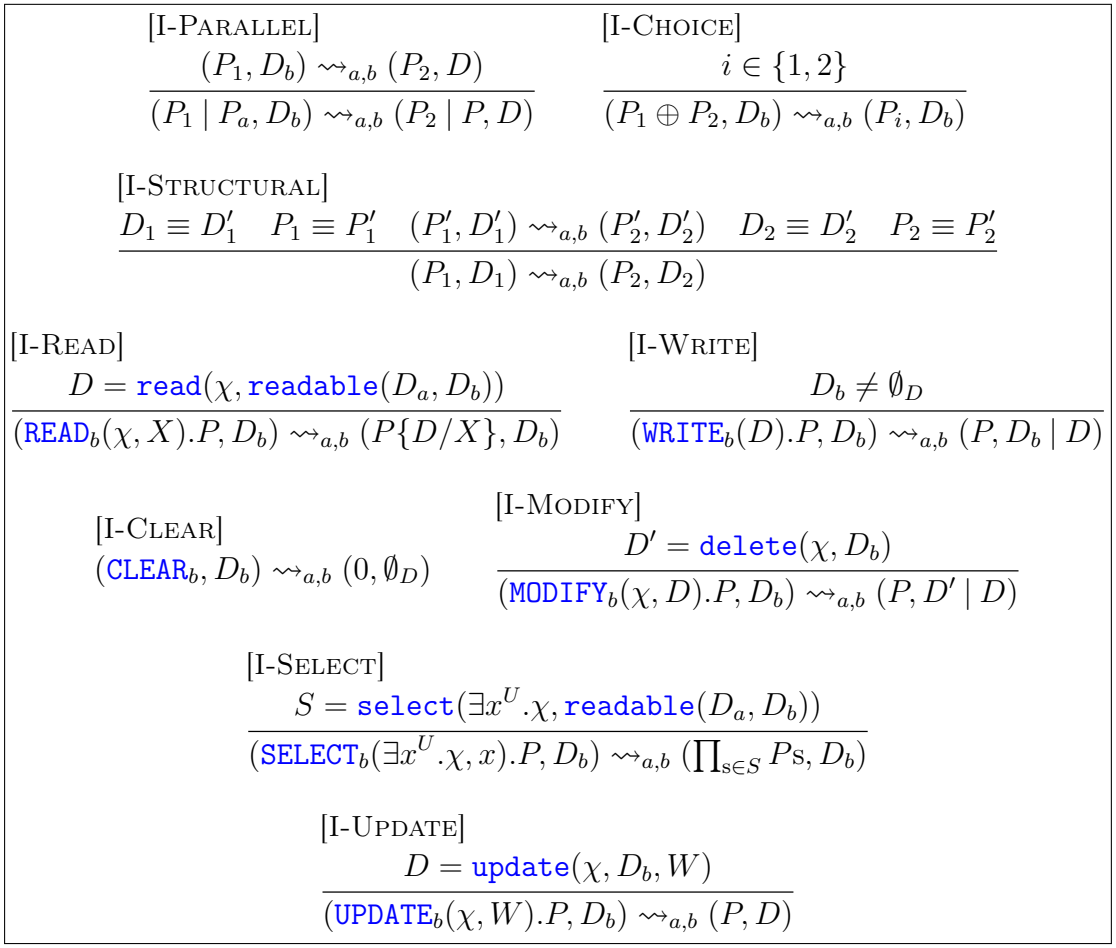

Figure 4.10: Interaction rules

In rule [I-MODIFY], the data D' are obtained from D_b by deleting the triples that satisfy the pattern χ and they, together with D in parallel, replace D_b. This rule enables users to unblock a blocked user.

In rule [I-SELECT], all the substitutions belonging to the set S are applied to process P and then those processes run as a parallel composition. For each triple in the data D_b, accessible to D_a, that satisfies the pattern $\exists x^U.\chi$ there is a substitution in the set S.

In rule [I-UPDATE], the data D are obtained from D_b by changing privacy protection policies on all triples that satisfy the pattern χ and they replace D_b after the interaction. The new privacy protection policies are set to be W on all the triples identified by χ.

We write $(P, D) \rightsquigarrow_{a,b}$ if $(P, D) \rightsquigarrow_{a,b} (P', D')$ for some P' and D' and $(P, D) \not\rightsquigarrow_{a,b}$ if not $(P, D) \rightsquigarrow_{a,b}$.

Example 4.2.11. We now specify policies for the second process from Example 4.2.2 and give examples of this process interacting with data $D_{\texttt{Alice}}$ from Example 4.2.1. We use the results of auxiliary functions from Example 4.2.10 and denote by P the following process

$$\texttt{SELECT}_{\texttt{Alice}}(\exists x^{U_{\texttt{Alice}}}.(x, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}}, x).\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(x, y, z)^{U_{\texttt{Alice}}}, \emptyset_D)$$

In case this process interacts on behalf of user `Bob`, after the interaction it becomes 0 and the data $D_{\texttt{Alice}}$ stays unchanged.

$$\frac{\emptyset = \texttt{select}(\exists x^{U_{\texttt{Alice}}}.(x, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}}, \texttt{readable}(D_{\texttt{Bob}}, D_{\texttt{Alice}}))}{(P, D_{\texttt{Alice}}) \rightsquigarrow_{\texttt{Bob,Alice}} (0, D_{\texttt{Alice}})}$$

In case the process P interacts on behalf of user `Alice`, the function `select` provides a substitution, so, in two steps, the process becomes 0 and the data $D_{\texttt{Alice}}$ gets to be changed.

$$\frac{\{\{\texttt{Alice}/x\}\} = \texttt{select}(\exists x^{U_{\texttt{Alice}}}.(x, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}}, \texttt{readable}(D_{\texttt{Alice}}, D_{\texttt{Alice}}))}{(P, D_{\texttt{Alice}}) \rightsquigarrow_{\texttt{Alice,Alice}} (\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, \emptyset_D), D_{\texttt{Alice}})}$$

$$\frac{D = \texttt{delete}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, D_{\texttt{Alice}})}{(\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, \emptyset_D), D_{\texttt{Alice}}) \rightsquigarrow_{\texttt{Alice,Alice}} (0, D)}$$

The third process from Example 4.2.2, using [I-WRITE] adds a data triple to the data of user `Alice`.

$$\frac{D_{\texttt{Alice}} \neq \emptyset_D}{\begin{array}{l}(\texttt{WRITE}_{\texttt{Alice}}((\texttt{photo}, \texttt{at}, \texttt{party})^U), D_{\texttt{Alice}}) \\ \rightsquigarrow_{\texttt{Bob,Alice}} (0, D_{\texttt{Alice}} \mid (\texttt{photo}, \texttt{at}, \texttt{party})^U)\end{array}}$$

△

Reduction relation is the smallest relation between networks defined by the rules in Figure 4.11. Interaction relation enables us to define in an elegant way

$$\text{[R-User]}\ \frac{(P_a, D_b) \rightsquigarrow_{a,b} (P'_a, D'_b) \quad a \neq b}{a[D_a \parallel P] \,[]\, b[D_b \parallel P_b] \to a[D_a \parallel P'_a] \,[]\, b[D'_b \parallel P_b]} \qquad \text{[R-Self]}\ \frac{(P_a, D_a) \rightsquigarrow_{a,a} (P'_a, D'_a)}{a[D_a \parallel P_a] \to a[D'_a \parallel P'_a]}$$

$$\text{[R-Struct]}\ \frac{N_1 \equiv N'_1 \quad N'_1 \to N'_2 \quad N_2 \equiv N'_2}{N_1 \to N_2} \qquad \text{[R-Parallel]}\ \frac{N_1 \to N_2}{N_1 \,[]\, N \to N_2 \,[]\, N}$$

Figure 4.11: Reduction relation

the reduction rules that distinguish between interaction of different users and self-interactions. Rule [R-User] describes interaction between two users with different names, while rule [R-Self] describes self-interaction between user's process and its own data. Rules [R-Struct] and [R-Parallel] are standard rules which formalize the property that the reduction relation is defined up to structural congruence and it is closed with respect to contexts.

We use $\to^*$ to denote the reflexive and transitive closure of $\to$.

Example 4.2.12. We now use some results shown in Examples 4.2.10 and 4.2.11. Since we know that

$$\begin{array}{rcl} D & = & \texttt{delete}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, D_{\texttt{Alice}}) \\ & = & (\texttt{UNS}, \texttt{organizes}, \texttt{event})^{U_{\texttt{P}}} \\ & & |(\texttt{Bob}, \texttt{organizes}, \texttt{event})^{U_{\texttt{P}}} \\ & & |(\texttt{paper}, \texttt{published}, \texttt{journal})^{U_{\texttt{Alice}} \vee \exists x.(x, \texttt{is}, \texttt{researcher})} \end{array}$$

We apply reduction rule [R-Self] twice in order to derive

$$\texttt{Alice}[D_{\texttt{Alice}} \parallel P] \to^* \texttt{Alice}[D \parallel 0].$$

△

4.2.4 Well-behaved networks

We say that an user owns data if the name enclosing the data is accessible to that user. We also consider a blocked user as the owner of data enclosed under her name. Informally, in a well-behaved network:

- the data of each active user must satisfy the policy of its own name and implicitly such user can always completely read its own data;
- a user with the data that satisfy the privacy policy of another user's name, must also have access to all the data triples belonging to the user with that name;
- a blocked user does not have access to any data and implicitly cannot read and select data from the network;
- a blocked user cannot be activated by writing data.

- an active user is allowed to modify, update or delete the data only of another user that he owns.
- a blocked user is allowed to change the data of another user only if the privacy policy of the blocked user is stronger than the policy of the other user. This enables blocked users to finish changes to the network they already started, but, as already stated, they cannot read nor select data.

The following definition formalizes what we consider as well-behaved network, i.e. when we consider that privacy of data is completely protected.

Definition 4.2.13. *Let $N \rightarrow^* a[D_a \parallel P_a] \,[]\, N'$ and $\mathcal{P}(a) = U$. We say that N is well behaved if the following three assertions hold:*

(1) if $D_a \equiv (a_1, a_2, a_3)^W \mid E$, then

(i) $D_a \models \texttt{ASK}(U)$;

(ii) $D \models \texttt{ASK}(U)$ implies $D \models \texttt{ASK}(W)$;

(2) if $D_a = \emptyset_D$ and $c \in \mathsf{IRIs}$, then

(i) $\texttt{readable}(D_a, D) = \emptyset_D$;

(ii) $(\texttt{WRITE}_a(D), D_a) \not\rightarrow_{c,a}$;

(3) Let $N' \equiv b[D_b \parallel P_b] \,[]\, N''$ and $\mathcal{P}(b) = V$. If

- *$P_a \equiv \texttt{MODIFY}_b(\chi, D).P \mid Q$, or*
- *$P_a \equiv \texttt{UPDATE}_b(\chi, V).P \mid Q$, or*
- *$P_a \equiv \texttt{CLEAR}_b \mid Q$,*

then

(i) if $D_a \neq \emptyset_D$, then $D_a \models \texttt{ASK}(V)$;

(ii) if $D_a = \emptyset_D$, then $D \models \texttt{ASK}(U)$ implies $D \models \texttt{ASK}(V)$.

The following two theorems show that, in a well-behaved network, an owner of the data fully access its own data.

Theorem 4.2.14. *Let $N \equiv a[D_a \parallel P_a] \mid N'$ be a well-behaved network. Then $\texttt{readable}(D_a, D_a) = D_a$.*

Proof. Let $\mathcal{P}(a) = U$. The proof is by induction on the structure of D_a.

$\boxed{D_a = \emptyset_D}$ By the definition of `readable`, given in Figure 4.9, we know that $\texttt{readable}(D_a, D_a) = \texttt{readable}(D_a, \emptyset_D) = \emptyset_D = D_a$.

$\boxed{D_a = (a_1, a_2, a_3)^W}$ From (ii) of Definition 4.2.13 we know that $D_a \models \texttt{ASK}(U)$. Then, from (i) of the same definition we obtain $D_a \models \texttt{ASK}(W)$. We conclude $\texttt{readable}(D_a, D_a) = \texttt{readable}(D_a, (a_1, a_2, a_3)^W) = (a_1, a_2, a_3)^W = D_a$ by the definition of function `readable`.

$\boxed{D_a = D_1 \mid D_2}$ In this case $\texttt{readable}(D_a, D_a) = \texttt{readable}(D_a, D_1 \mid D_2) = \texttt{readable}(D_a, D_1) | \texttt{readable}(D_a, D_2)$ by the definition of function `readable`. By induction hypothesis we obtain $\texttt{readable}(D_a, D_1) = D_1$ and $\texttt{readable}(D_a, D_2) = D_2$, which implies $\texttt{readable}(D_a, D_1 \mid D_2) = D_1 \mid D_2 = D_a$. □

Theorem 4.2.15. *Let $N \equiv a[D_a \parallel P_a] \mid b[D_b \parallel P_b] \mid N'$ be a well-behaved network and $\mathcal{P}(b) = V$. Then, $D_a \models \texttt{ASK}(V)$ implies $\texttt{readable}(D_a, D_b) = D_b$.*

Proof. Let $\mathcal{P}(a) = U$. The proof is by induction on the structure of D_b.

$\boxed{D_b = \emptyset_D}$ By the definition of `readable`, given in Figure 4.9, we know that $\texttt{readable}(D_a, D_b) = \texttt{readable}(D_a, \emptyset_D) = \emptyset_D = D_b$.

$\boxed{D_b = (a_1, a_2, a_3)^W}$ From $D_b \models \texttt{ASK}(V)$ and from (i) of Definition 4.2.13 we obtain $D_a \models \texttt{ASK}(W)$. We conclude $\texttt{readable}(D_a, D_b) = \texttt{readable}(D_a, (a_1, a_2, a_3)^W) = (a_1, a_2, a_3)^W = D_b$ by the definition of function `readable`.

$\boxed{D_b = D_1 \mid D_2}$ In this case $\texttt{readable}(D_a, D_b) = \texttt{readable}(D_a, D_1 \mid D_2) = \texttt{readable}(D_a, D_1) | \texttt{readable}(D_a, D_2)$ by the definition of function `readable`. By induction hypothesis we obtain $\texttt{readable}(D_a, D_1) = D_1$ and $\texttt{readable}(D_a, D_2) = D_2$, which implies $\texttt{readable}(D_a, D_1 \mid D_2) = D_1 \mid D_2 = D_b$. □

Here are few examples of ill-behaved users and networks to illustrate what kind of errors we want to eliminate with the type system presented in the following section.

- If $D_{\texttt{Alice}}$ would contain a triple $(\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}}$ a user named `Alice` would violate condition (1).(ii) of Definition 4.2.13. That is because, for example,

  (a) $(\texttt{Alice}, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}} \models \texttt{ASK}(U_{\texttt{Alice}})$

  while it does not hold

  $$(\texttt{Alice}, \texttt{is}, \texttt{person})^{U_{\texttt{Alice}}} \models \texttt{ASK}(U_{\texttt{Bob}}).$$

  Let us denote

  $$D'_{\texttt{Alice}} = D_{\texttt{Alice}} \mid (\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}}.$$

  Notice that condition (1).(i) is satisfied since from (a) we can derive $D'_{\texttt{Alice}} \models \texttt{ASK}(U_{\texttt{Alice}})$ by Definition 4.2.3.

- Obviously, the process

  $$\texttt{WRITE}_{\texttt{Alice}}((\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}})$$

  which aims to write the just mentioned triple to the data of user named `Alice`, i.e. to add it to $D_{\texttt{Alice}}$, must be considered as unsafe.

- The process
$$\texttt{WRITE}_{\texttt{Bob}}((\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}})$$
should be considered safe because it aims to add a triple to the data of the user named Bob with the exact privacy protection policy of that user. Let us denote
$$D'_{\texttt{Bob}} = (\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}} \mid D_{\texttt{Bob}}.$$
It does not hold
$$D'_{\texttt{Bob}} \models \texttt{ASK}(U_{\texttt{Bob}})$$
This error is induced by the fact that the data of user Bob violated condition (1).(i) in first place. Namely, it does not hold
$$D_{\texttt{Bob}} \models \texttt{ASK}(U_{\texttt{Bob}}).$$
Notice that
$$\texttt{readable}(D_{\texttt{Bob}}, D_{\texttt{Bob}}) = D_{\texttt{Bob}}$$
and that (1).(ii) holds for user Bob.

- The process
$$\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, \emptyset_D)$$
deletes triples from $D_{\texttt{Alice}}$ and puts $\emptyset_D$ instead of these triples, i.e. it deletes triples from $D_{\texttt{Alice}}$. For the data D of the user Alice computed during the reduction, as in Example 4.2.12, the condition (1).(i) does not hold.

- The process
$$\texttt{UPDATE}_{\texttt{Alice}}((\texttt{Alice}, \texttt{has affiliation}, \texttt{UNS})^{U_{\texttt{Alice}}}, U_{\texttt{Bob}})$$
would make the triple $(\texttt{Alice}, \texttt{has affiliation}, \texttt{UNS})^{U_{\texttt{Alice}}}$ inaccessible to its owner, Alice.

- Let us consider a situation in which the data of the user Bob satisfy condition (1).(i). These data, denoted by $D'_{\texttt{Bob}}$, must be such that they contain the triple $(\texttt{Bob}, \texttt{is}, \texttt{person})^{W}$, for example
$$D'_{\texttt{Bob}} = D_{\texttt{Bob}} \mid (\texttt{Bob}, \texttt{is}, \texttt{person})^{U_{\texttt{P}}}.$$
The process running on behalf of user Alice
$$\texttt{CLEAR}_{\texttt{Bob}}$$
violates condition (3).(i) Notice that we consider this process an error, although it holds
$$\texttt{readable}(D_{\texttt{Alice}}, D'_{\texttt{Bob}}) = D'_{\texttt{Bob}}.$$
In this specific case, user Alice can steel the *identity (triples that actually satisfy the policy)* of user Bob by running the following process
$$P_{\texttt{Alice}} = \texttt{READ}_{\texttt{Bob}}((\texttt{Bob}, \texttt{is}, \texttt{person})^{U_{\texttt{P}}}, X).\texttt{WRITE}_{\texttt{Alice}}(X),$$
which would allow Alice behave as Bob in the future actions. On the other hand, the user Bob cannot do such a thing to the user Alice.

Some other choices could be made in the definition of well-behaved network, but we have opted for the current one, since it satisfactorily describes privacy properties. In particular, in condition (3) we could consider more processes to be allowed to delete, modify and update data or just parts of data. The proposed option is meaningful with the respect to the privacy protection. Furthermore, it would be possible to strengthen condition (1) by adding the same condition for policies of triples and their elements. Namely, we could assume that a user that is allowed to access a triple must be allowed to access all its elements. This would imply that any name mentioned as an element of a triple belonging to an user must be owned by that user. This approach is similar to the one proposed in [127], which is in our case too restrictive.

4.3 Type system

Aiming to prevent privacy violations, we introduce a type system that enforces well-behavedness of networks. In Section 4.3.1 we introduce an order on privacy policies, which is the main tool for the type assignment system presented in Sections 4.3.2, 4.3.3 and 4.3.4.

4.3.1 Policy comparison

We introduce a relation on privacy protection policies which will enable us to compare them.

Definition 4.3.1 (Policy comparison relation). *We define a partial order $\preccurlyeq$ on privacy policies by the following rules:*

$$\text{[C-OrL]}\quad U \preccurlyeq U \vee V \qquad \text{[C-OrR]}\quad V \preccurlyeq U \vee V \qquad \text{[C-Subst]}\quad U\{a/x\} \preccurlyeq \exists x.U$$

$$\text{[C-Or]}\quad \frac{U \preccurlyeq W \quad V \preccurlyeq W}{U \vee V \preccurlyeq W} \qquad \text{[C-Exists]}\quad \frac{U\{a/x\} \preccurlyeq V\{a/y\}}{\exists x.U \preccurlyeq \exists y.V}$$

If $U \preccurlyeq V$ we say that U is more or equally restrictive *than V or, equivalently, that V is* less or equally restrictive *than U.*

Using this partial order the type system will ensure that the privacy policy of a triple is bounded below by the privacy policies of the user name it is enclosed with. We now prove that a user which is allowed to access a resource protected with a policy is allowed to access any resource protected with a less restrictive policy.

Theorem 4.3.2 (Policy satisfaction). *If $D \models \texttt{ASK}(U)$ and $U \preccurlyeq V$, then $D \models \texttt{ASK}(V)$.*

Proof. The proof is by induction on the derivation of $U \preccurlyeq V$.

[C-OrL] In this case $V = U \vee V'$. From $D \models \texttt{ASK}(U)$, Lemma 4.2.4, [A-OrL] and [A-Ask] we obtain $D \models \texttt{ASK}(V)$.

[C-ORR] In this case $V = V' \vee U$. From $D \models \texttt{ASK}(U)$, Lemma 4.2.4, [A-ORR] and [A-ASK] we obtain $D \models \texttt{ASK}(V)$.

[C-OR] In this case $U = U' \vee V'$ and $U' \preccurlyeq V$ and $V' \preccurlyeq V$. From $D \models \texttt{ASK}(U)$, Lemma 4.2.4 and rules [A-ORL] and [A-ORR] we know that $D \models \texttt{ASK}(U')$ or $D \models \texttt{ASK}(V')$. In both cases we derive a proof of $D \models \texttt{ASK}(V)$ by induction hypothesis.

[C-SUBST] In this case $U = U'\{a/x\}$ and $V = \exists x.U'$. From $D \models \texttt{ASK}(U)$, Lemma 4.2.4, [A-EXISTS] and [A-ASK] we obtain $D \models \texttt{ASK}(V)$.

[C-EXISTS] In this case $U = \exists x.U'$ and $V = \exists y.V'$. From $D \models \texttt{ASK}(U)$, Lemma 4.2.4, [A-EXISTS] and [A-ASK] we obtain $D \models \texttt{ASK}(U'\{a/x\})$. From $U \preccurlyeq V$ by Definition 4.3.1 we have $U'\{a/x\} \preccurlyeq V'\{a/x\}$ and then by induction hypothesis we obtain $D \models \texttt{ASK}(V'\{a/x\})$. Finally, from Lemma 4.2.4, [A-EXISTS] and [A-ASK] we conclude $D \models V$. □

Intuitively, the set of data that satisfies a privacy policy U, that is more restrictive than another one, is the subset of the set of data that satisfy the policy U. For example, only processes that act on behalf of users whose data contain the triple $(\texttt{Bob}, \texttt{is}, \texttt{person})^U$ are allowed to access a triple or name protected with privacy policy $(\texttt{Bob}, \texttt{is}, \texttt{person})$, while in order to access a triple or name protected with policy $(\texttt{Bob}, \texttt{is}, \texttt{person}) \vee (\texttt{Alice}, \texttt{is}, \texttt{person})$ a process must act on behalf of a user whose data contain the triple $(\texttt{Bob}, \texttt{is}, \texttt{person})^U$ or the triple $(\texttt{Alice}, \texttt{is}, \texttt{person})^V$. Similarly, in order to access a triple or name protected with policy $\exists x.(x, \texttt{is}, \texttt{person})$ a process must act on behalf of a user whose data contain a triple with an arbitrary subject, predicate `is` and object `person`. We can derive, by rule [C-ORL]

$$(\texttt{Bob}, \texttt{is}, \texttt{person}) \preccurlyeq (\texttt{Bob}, \texttt{is}, \texttt{person}) \vee (\texttt{Alice}, \texttt{is}, \texttt{person})$$

and by rule [C-EXISTS]

$$(\texttt{Bob}, \texttt{is}, \texttt{person}) \preccurlyeq \exists x.(x, \texttt{is}, \texttt{person})$$

and by rule [C-OR]

$$(\texttt{Bob}, \texttt{is}, \texttt{person}) \vee (\texttt{Alice}, \texttt{is}, \texttt{person}) \preccurlyeq \exists x.(x, \texttt{is}, \texttt{person}).$$

By Theorem 4.3.2, if $D \models \texttt{ASK}((\texttt{Bob}, \texttt{is}, \texttt{person}))$, then

$$D \models \texttt{ASK}((\texttt{Bob}, \texttt{is}, \texttt{person}) \vee (\texttt{Alice}, \texttt{is}, \texttt{person}))$$

and

$$D \models \texttt{ASK}(\exists x.(x, \texttt{is}, \texttt{person})).$$

4.3.2 Types

We distinguish five kinds of types, given by the syntax in Figure 4.12. The type $\mathit{Name}(U)$ denotes the type of a name which has the privacy protection policy U.

$Name(U)$	name type	$Data(U)$	data type
$Pattern(U)$	pattern type	$Process(U)$	process type
$Network$	network type		

Figure 4.12: Syntax of the types.

The data whose all triples and all triples' components have policies which are not more restrictive than U have type $Data(U)$. The data identified by the pattern with the type $Pattern(U)$ have the type $Data(V)$, where V is at most restrictive as U. A process with the type $Process(U)$ does not change, delete or update data of users whose names have policies more restrictive than U. $Network$ denotes the type of the network.

A *type environment* Γ associates name and data variables with name and data types, i.e. we define:

$$\Gamma ::= \emptyset \mid \Gamma, x : Name(U) \mid \Gamma, X : Data(U).$$

We denote by $dom(\Gamma)$ the set of all name and data variables that appear in Γ. For a well-formed environment Γ, we say that $\Gamma, x : Name(U)$ is well formed if $x \notin dom(\Gamma)$, and similarly, $\Gamma, X : Data(U)$ is well formed if $X \notin dom(\Gamma)$.

We use the environment by the standard axioms:

[T-Name Variable] $\Gamma, x : Name(U) \vdash x : Name(U)$

[T-Data Variable] $\Gamma, X : Data(U) \vdash X : Data(U)$

The five kinds of types induce five forms of type judgments:

$$\Gamma \vdash u : Name(U) \quad \Gamma \vdash D : Data(U) \quad \Gamma \vdash \chi : Pattern(U)$$

$$\Gamma \vdash P : Process(U) \quad \Gamma \vdash N : Network.$$

The first four state that the name u, the data D, the pattern χ and the process P are well typed in the environment Γ and for the privacy protection policy U. The fifth judgment states that the network N is well typed in Γ.

4.3.3 Typing names, data and patterns

Typing rules for names, data and patterns, given in Figure 4.13 and Figure 4.14, are described in the following paragraphs.

Rule [T-Name] states that if a privacy protection policy is assigned to a name by the function $\mathcal{P}(\cdot)$, then that name is well typed in any environment for the assigned policy.

Rule [T-Empty Data] states that empty data is well typed in any environment and for any privacy protection policy.

Rule [T-Data Triple] states that a data triple with the privacy policy U is well typed for policies that are more or equally restrictive than U. That is expressed by the relation $V \preccurlyeq U$.

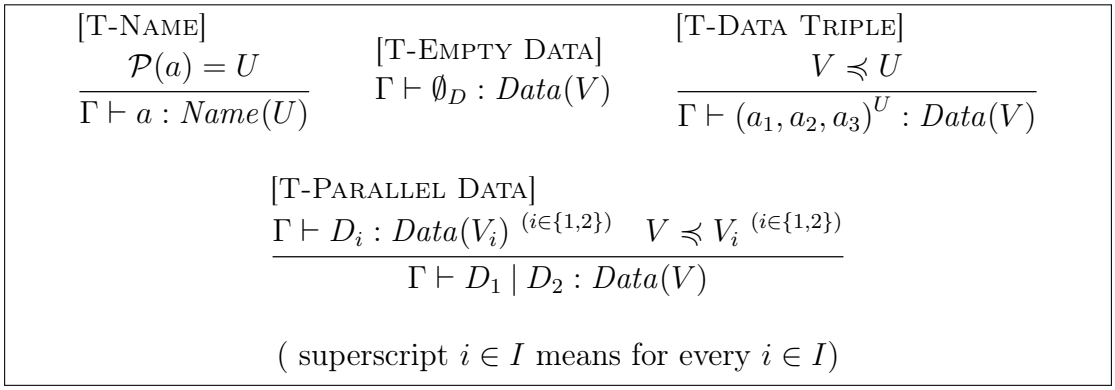

Figure 4.13: Typing rules for names and data

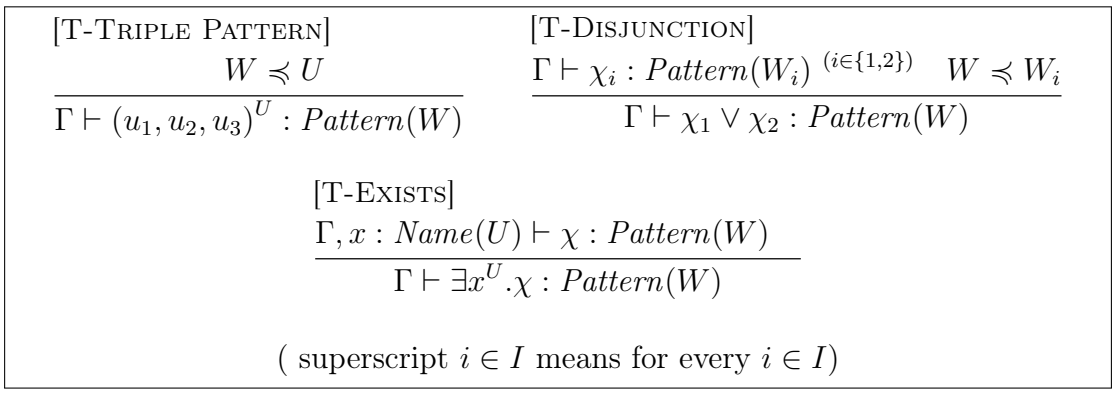

Figure 4.14: Typing rules for patterns

Rule [T-Parallel Data] states that a parallel composition of data is well typed for a privacy protection policy if the composed data have less or equally restrictive privacy protection policies than the specified policy.

Typing rules for the patterns are such that they ensure that the data identified by a pattern is well typed for a policy which is at most restrictive as the policy for which the pattern is well typed.

Rule [T-Triple Pattern] is essentially the same as [T-Data Triple].

Rule [T-Disjunction] states that a disjunction pattern is well typed in an environment and for a policy if both disjuncts are well typed in the same environment and for a policy which is less or equally restrictive than the specified policy.

Rule [T-Exists] states that an existential pattern $\exists x^U.\chi$ is well typed in an environment and for a policy if the pattern χ is well typed in the same environment extended with the type assignment which assigns policy U to the variable x.

4.3.4 Typing processes and networks

Typing rules for processes, given in Figure 4.15, are described in the following paragraphs.

Rule [T-Inaction] states that the terminating process is well typed in any environment and for any privacy protection policy.

Rules [T-Choice] and [T-Parallel Process] state that choice and parallel composition of two processes are well typed in an environment and for a privacy protection policy if both processes are well typed in this environment and for the privacy protection policy.

Rule [T-Replication] states that a replication of a process is well typed in an environment and for a privacy protection policy if the process is well typed in this environment and for this privacy protection policy.

Rule [T-Read] states that the read process $\texttt{READ}_u(\chi, X).P$ is well typed in an environment for a privacy protection policy U if the pattern χ is well typed in the same environment and for a policy W and the continuation process P is well typed in the environment extended with $X : \mathit{Data}(W)$ for the policy U. From Lemma 4.4.8, we know that any data identified by the pattern will be well typed in the same environment and the for the same policy as the pattern.

Rule [T-Select] states that the select process $\texttt{SELECT}_u(\exists x^W.\chi, x).P$ is well typed in an environment and for a privacy protection policy if pattern χ and its continuation process P is well typed in the environment extended with $x : \mathit{Name}(W)$. From Lemma 4.4.9, we know that any selected name will have the appropriate type. Unlike in the previous rule, here we are not interested in the type of the pattern, since we only need the information about the type of the binding variable.

Rule [T-Write] states that a process that aims to write data δ to a user well typed for V is well typed in an environment and for a privacy protection policy U, if V is more or equally restrictive than W, where W is the policy for which δ is well typed, and the continuation of the process is well typed for U. This ensures that users that have access to the whole data will still have that privilege after their data is extended with new data.

$$\text{[T-Inaction]}\quad \Gamma \vdash 0 : \mathit{Process}(U) \qquad \text{[T-Choice]}\quad \frac{\Gamma \vdash P_i : \mathit{Process}(U)^{\ (i\in\{1,2\})}}{\Gamma \vdash P_1 \oplus P_2 : \mathit{Process}(U)}$$

$$\text{[T-Parallel Process]}\quad \frac{\Gamma \vdash P_i : \mathit{Process}(U)^{\ (i\in\{1,2\})}}{\Gamma \vdash P_1 \mid P_2 : \mathit{Process}(U)} \qquad \text{[T-Replication]}\quad \frac{\Gamma \vdash P : \mathit{Process}(U)}{\Gamma \vdash *P : \mathit{Process}(U)}$$

$$\text{[T-Read]}\quad \frac{\Gamma \vdash \chi : \mathit{Pattern}(W) \quad \Gamma, X : \mathit{Data}(W) \vdash P : \mathit{Process}(U)}{\Gamma \vdash \mathtt{READ}_u(\chi, X).P : \mathit{Process}(U)}$$

$$\text{[T-Select]}\quad \frac{\Gamma, x : \mathit{Name}(W) \vdash \chi : \mathit{Pattern}(V) \qquad \Gamma, x : \mathit{Name}(W) \vdash P : \mathit{Process}(U)}{\Gamma \vdash \mathtt{SELECT}_u(\exists x^W.\chi, x).P : \mathit{Process}(U)}$$

$$\text{[T-Write]}\quad \frac{\Gamma \vdash u : \mathit{Name}(V) \quad \Gamma \vdash \delta : \mathit{Data}(W) \quad \Gamma \vdash P : \mathit{Process}(U) \quad V \preccurlyeq W}{\Gamma \vdash \mathtt{WRITE}_u(\delta).P : \mathit{Process}(U)}$$

$$\text{[T-Clear]}\quad \frac{\Gamma \vdash u : \mathit{Name}(V) \quad U \preccurlyeq V}{\Gamma \vdash \mathtt{CLEAR}_u : \mathit{Process}(U)}$$

$$\text{[T-Modify]}\quad \frac{\Gamma \vdash u : \mathit{Name}(V) \quad \Gamma \vdash P : \mathit{Process}(U) \quad U \preccurlyeq V \quad \Gamma \vdash \chi : \mathit{Pattern}(W) \quad \Gamma \vdash D : \mathit{Data}(V) \quad D \models \mathtt{ASK}(V)}{\Gamma \vdash \mathtt{MODIFY}_u(\chi, D).P : \mathit{Process}(U)}$$

$$\text{[T-Update]}\quad \frac{\Gamma \vdash u : \mathit{Name}(V) \quad \Gamma \vdash \chi : \mathit{Pattern}(W) \quad \Gamma \vdash P : \mathit{Process}(U) \quad U \preccurlyeq V \preccurlyeq W}{\Gamma \vdash \mathtt{UPDATE}_u(\chi, W).P : \mathit{Process}(U)}$$

(superscript $i \in I$ means for every $i \in I$)

Figure 4.15: Typing rules for processes

[T-User]
$$\frac{\emptyset \vdash a : Name(U) \quad \emptyset \vdash D : Data(U) \quad \emptyset \vdash P : Process(U) \quad D \models \texttt{ASK}(U)}{\emptyset \vdash a[D \parallel P] : Network}$$

[T-Blocked]
$$\frac{\emptyset \vdash a : Name(U) \quad \emptyset \vdash P : Process(U)}{\emptyset \vdash a[\emptyset_D \parallel P] : Network}$$

[T-Network]
$$\frac{\emptyset \vdash N_i : Network \ ^{(i \in \{1,2\})}}{\emptyset \vdash N_1 \,[]\, N_2 : Network}$$

(superscript $i \in I$ means for every $i \in I$)

Figure 4.16: Typing rules for networks

Rule [T-Clear] states that a process aiming to delete the whole data of a user named u is well typed for a privacy protection policy, if the policy is more or equally restrictive than the policy for which name u is well typed. This rule will ensure that the data of a user can be completely deleted only by an user that owns the data.

Rule [T-Modify] states that a process aiming to replace data (identified by the pattern χ) of a user u by D, is well typed for a privacy protection policy U, if policy U is more or equally restrictive than the policy V, for which u is well typed. The new data D must be well typed for V and satisfy V. The continuation process must be well typed for U. The premise $D \models \texttt{ASK}(V)$ ensures that the original owner of the changed data does not get blocked and that data $D \neq \emptyset_D$, i.e. this process cannot just delete data. The same premise ensures that, if pattern χ deletes all triples that build the identity of the user u, D contains some new triples that satisfy the policy V. The premise $\Gamma \vdash D : Data(V)$ ensures that the user u is allowed to access the new data. Notice that data which are added to a user by modifying must satisfy the policy of the user, while triples that are written to a user should just have less restrictive policy than the policy of the user.

Rule [T-Update] states that a process aiming to update privacy policies on the data of the user named u is well typed for a privacy protection policy U if $U \preccurlyeq V \preccurlyeq W$, where V is the privacy policy of u and W is the new privacy policy. The continuation is well typed for U. The premise $U \preccurlyeq V \preccurlyeq W$ ensures that only owners of the data may update them, and that newly obtained data stay available to the original owner.

Typing rules for networks, given in Figure 4.16, are described as follows.

Rule [T-User] states that a user with enclosed data and process is well typed, if the user name, the data and the process are well typed for the same privacy protection policy. The data must satisfy the privacy policy of the enclosing name. A straightforward consequence of the type assignment rules is that if a user $a[D \parallel P]$ is well typed, then both the data D and the process P do not contain occurrences of free variables.

Rule [T-Blocked] states that a user with empty data is well typed if the enclosed process is well typed for the same policy as the name of the user. We call such user *blocked user* since from Lemma 4.2.5 we know that such user is not

allowed to access any data.

Rule [T-Parallel Network] states that the parallel composition of well-typed networks is a well typed network too.

Example 4.3.3. Again we recall Example 4.2.1. We can now show that the examples given in Section 4.2.4 are not typeable. Namely, user `Alice` holding the data

$$D'_{\texttt{Alice}} = D_{\texttt{Alice}} \mid (\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}}$$

and user `Bob` holding the process

$$P_{\texttt{Bob}} = \texttt{WRITE}_{\texttt{Alice}}((\texttt{photo}, \texttt{at}, \texttt{party})^{U_{\texttt{Bob}}})$$

are rejected by the type system because $U_{\texttt{Alice}} \not\preccurlyeq U_{\texttt{Bob}}$. We are not able to type user `Bob` holding data $D_{\texttt{Bob}}$ because $D_{\texttt{Bob}} \not\models \texttt{ASK}(U_{\texttt{Bob}})$. The process

$$\texttt{MODIFY}_{\texttt{Alice}}(\exists y^{U_{\texttt{P}}}.\exists z^{U_{\texttt{P}}}.(\texttt{Alice}, y, z)^{U_{\texttt{Alice}}}, \emptyset_D)$$

is rejected because $\emptyset_D \not\models \texttt{ASK}(U_{\texttt{Alice}})$. The process

$$\texttt{UPDATE}_{\texttt{Alice}}((\texttt{Alice}, \texttt{has affiliation}, \texttt{UNS})^{U_{\texttt{Alice}}}, U_{\texttt{Bob}})$$

is rejected because $U_{\texttt{Alice}} \not\preccurlyeq U_{\texttt{Bob}}$. Finally, the process

$$P_{\texttt{Alice}} = \texttt{CLEAR}_{\texttt{Bob}}$$

is rejected because $U_{\texttt{Alice}} \not\preccurlyeq U_{\texttt{Bob}}$. On the other hand, it is easy to check that

$$\emptyset \vdash \texttt{READ}_{\texttt{Bob}}((\texttt{Bob}, \texttt{is}, \texttt{person})^{U_{\texttt{P}}}, X).\texttt{WRITE}_{\texttt{Alice}}(X) : \mathit{Process}(U_{\texttt{Alice}})$$

holds. △

4.4 Type soundness

In the current section, we prove that the proposed type system guarantees the operational property of type preservation under reduction: all networks obtained by reduction starting from a well-typed network are again well-typed. Then, we prove that the type system is safe: a well-typed network is well behaved. By proving these two properties, we conclude that our type system is sound.

Before we start with proofs of the two main results we need to prove several auxiliary lemmas. The following two lemmas are standard and they say that typing is preserved by structural congruence and by substitutions.

Lemma 4.4.1.

1. *If* $\emptyset \vdash D : \mathit{Data}(U)$ *and* $D \equiv D'$, *then* $\emptyset \vdash D' : \mathit{Data}(U)$;
2. *If* $\emptyset \vdash P : \mathit{Process}(U)$ *and* $P \equiv P'$, *then* $\emptyset \vdash P' : \mathit{Process}(U)$;
3. *If* $\emptyset \vdash N : \mathit{Network}$ *and* $N \equiv N'$, *then* $\emptyset \vdash N' : \mathit{Network}$.

Proof. Straightforward, by case analysis on the derivation of $D \equiv D'$, $P \equiv P'$ and $N \equiv N'$. □

Definition 4.4.2. *We define the substitution of a name in a pattern in such way that it does not affect privacy policies.*

$$\begin{array}{lcll} ((u_1, u_2, u_3)^U)\{a/x\} & = & (u_1, u_2, u_3)\{a/x\}^U & \\ (\chi \vee \psi)\{a/x\} & = & \chi\{a/x\} \vee \psi\{a/x\} & \\ (\exists y^U.\chi)\{a/x\} & = & \exists y^U.\chi\{a/x\} & \textit{if} \quad x \neq y \end{array}$$

Lemma 4.4.3 (Substitution).

1. *If* $\Gamma, x : Name(U) \vdash \chi : Pattern(V)$ *and* $\Gamma \vdash a : Name(U)$, *then* $\Gamma \vdash \chi\{a/x\} : Pattern(V)$.
2. *If* $\Gamma, x : Name(U) \vdash P : Process(V)$ *and* $\Gamma \vdash a : Name(U)$, *then* $\Gamma \vdash P\{a/x\} : Process(V)$.
3. *If* $\Gamma, X : Data(U) \vdash P : Process(V)$ *and* $\Gamma \vdash D : Data(U)$, *then* $\Gamma \vdash P\{D/X\} : Process(V)$.

Proof. 1. The proof is by induction on the derivation of $\Gamma, x : Name(U) \vdash \chi : Pattern(V)$.

[T-Triple Pattern] In this case:

- $\chi = (u_1, u_2, u_3)^W$
- (H.1) for every $i \in \{1, 2, 3\}$ it holds $\Gamma, x : Name(U) \vdash u_i : Name(U_i)$ and $V \preccurlyeq W$

In case $x \in \{u_1, u_2, u_3\}$, for example $x = u_1$, then by Definition 4.4.2, $\chi\{a/x\} = (a, u_2, u_3)^W$. From $\Gamma \vdash a : Name(U)$, (H.1) and [T- Triple Pattern] we obtain $\Gamma \vdash \chi\{a/x\} : Pattern(V)$.
In case $x \notin \{u_1, u_2, u_3\}$, then $\chi\{a/x\} = \chi$. From (H.1) and $x \notin \{u_1, u_2, u_3\}$ we obtain that for every $i \in \{1, 2, 3\}$ $\Gamma \vdash u_i : Name(U_i)$. We conclude by an application of rule [T-Triple Pattern].

[T-Disjunction] In this case:

- $\chi = \chi_1 \vee \chi_2$
- $\Gamma, x : Name(U) \vdash \chi_1 : Pattern(V_1)$ and $V \preccurlyeq V_1$
- $\Gamma, x : Name(U) \vdash \chi_2 : Pattern(V_2)$ and $V \preccurlyeq V_2$

We derive the proof by induction hypothesis, Definition 4.4.2 and rule [T-Disjunction].

[T-Exists] In this case:

- $\chi = \exists y^W.\chi'$

- (H.2) $\Gamma, x : \mathit{Name}(U), y : \mathit{Name}(W) \vdash \chi' : \mathit{Pattern}(V)$

(H.2) implies $y \notin \mathsf{dom}(\Gamma, x : \mathit{Name}(U))$, so we know $x \neq y$. Then, $\chi\{a/x\} = \exists y^W.(\chi'\{a/x\})$. From (H.2), $\Gamma \vdash a : \mathit{Name}(U)$ and induction hypothesis we obtain $\Gamma, y : \mathit{Name}(W) \vdash \chi'\{a/x\} : \mathit{Pattern}(V)$. We conclude the proof by an application of rule [T-Exists].

2. The proof is by induction on the derivation of $\Gamma, x : \mathit{Name}(U) \vdash P : \mathit{Process}(V)$. Most cases are very similar, we present few of them.

   [T-Select] In this case:

   - $P = \texttt{SELECT}_u(\exists y^W.\chi, y).P'$
   - (H.1) $\Gamma, x : \mathit{Name}(U), y : \mathit{Name}(W) \vdash \chi : \mathit{Pattern}(V')$
   - (H.2) $\Gamma, x : \mathit{Name}(U), y : \mathit{Name}(W) \vdash P' : \mathit{Process}(V)$

   From (H.1) we know that $x \neq y$. We give the proof in case $x = u$. Then, $P\{a/x\} = \texttt{SELECT}_a(\exists y^W.\chi\{a/x\}, y).P'\{a/x\}$. From (H.1), $\Gamma \vdash a : \mathit{Name}(U)$ (which implies $\Gamma, y : \mathit{Name}(W) \vdash a : \mathit{Name}(U)$) and item (1) of this lemma, we know that $\Gamma, y : \mathit{Name}(W) \vdash \chi\{a/x\} : \mathit{Pattern}(V')$. From (H.2) and $\Gamma \vdash a : \mathit{Name}(U)$, by induction hypothesis we obtain $\Gamma, y : \mathit{Name}(W) \vdash P'\{a/x\} : \mathit{Process}(V)$. We conclude the proof, in this case, with an application of rule [T-Select].

   [T-Parallel Process] In this case

   - $P = P_1 \mid P_2$
   - $\Gamma, x : \mathit{Name}(U) \vdash P_1 : \mathit{Process}(V)$
   - $\Gamma, x : \mathit{Name}(U) \vdash P_2 : \mathit{Process}(V)$

   We derive the proof by induction hypothesis and [T-Parallel Process].

   [T-Modify] In this case:

   - $P = \texttt{MODIFY}_u(\chi, D).P'$
   - $\Gamma, x : \mathit{Name}(U) \vdash u : \mathit{Name}(V')$
   - $\Gamma, x : \mathit{Name}(U) \vdash P' : \mathit{Process}(V)$
   - $V \preccurlyeq V'$
   - $\Gamma, x : \mathit{Name}(U) \vdash \chi : \mathit{Pattern}(W)$
   - $\Gamma, x : \mathit{Name}(U) \vdash D : \mathit{Data}(V')$
   - $D \models \texttt{ASK}(V')$

   If $x = u$, then $U = V'$ and $P\{a/x\} = \texttt{MODIFY}_a(\chi\{a/x\}, D).P'\{a/x\}$. Notice that $D\{a/x\} = D$ because the data triples do not contain variables. We derive the proof by induction hypothesis and [T-Modify]. If $x \neq u$, then $P\{a/x\} = \texttt{MODIFY}_u(\chi\{a/x\}, D).P'\{a/x\}$. From the typing rules for names

and data we can derive $\Gamma \vdash D : \mathit{Data}(V')$ and from the typing rules for names and name variables $\Gamma \vdash u : \mathit{Name}(V')$. Then, we obtain the proof by induction hypothesis and [T-Modify].

3. The proof is by induction on the derivation of $\Gamma, X : \mathit{Data}(U) \vdash P : \mathit{Process}(V)$.

   [T-Read] In this case:

   - $P = \mathtt{READ}_u(\chi, Y).P'$
   - $\Gamma, X : \mathit{Data}(U) \vdash \chi : \mathit{Pattern}(W)$
   - $\Gamma, X : \mathit{Data}(U), Y : \mathit{Data}(W) \vdash P' : \mathit{Process}(V)$

   From the typing rules for patterns we can derive $\Gamma \vdash \chi : \mathit{Pattern}(W)$, since patterns do not contain data variables. For the same reason $\chi\{D/X\} = \chi$. We derive the proof from item (1) of this lemma and induction hypothesis.

   [T-Write] In this case:

   - $P = \mathtt{WRITE}_u(\delta).P'$
   - $\Gamma, X : \mathit{Data}(U) \vdash u : \mathit{Name}(V')$
   - $\Gamma, X : \mathit{Data}(U) \vdash \delta : \mathit{Data}(W)$
   - $\Gamma, X : \mathit{Data}(U) \vdash P' : \mathit{Process}(V)$
   - $V' \preccurlyeq W$

   If $X = \delta$, then $U = W$ and $P\{D/X\} = \mathtt{WRITE}_u(D).P'\{D/X\}$. From the typing rules for names and name variables we can derive $\Gamma \vdash u : \mathit{Name}(V')$. We derive the proof by induction hypothesis and [T-Write]. If $X \neq \delta$, then $P\{D/X\} = \mathtt{WRITE}_u(\delta).P'\{D/X\}$. From the typing rules for names and name variables we can derive $\Gamma \vdash u : \mathit{Name}(V')$ and from the typing rules for data and data variables $\Gamma \vdash \delta : \mathit{Data}(W)$. We obtain the proof by induction hypothesis and [T-Write].

□

The next lemma shows that if the data are well typed for a policy, then they are well typed for any more or equally restrictive policy.

Lemma 4.4.4. *If* $\Gamma \vdash D : \mathit{Data}(V)$ *and* $W \preccurlyeq V$, *then* $\Gamma \vdash D : \mathit{Data}(W)$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash D : \mathit{Data}(V)$.

[T-Empty Data] In this case $D = \emptyset_D$. We derive the proof directly from rule [T-Empty Data].

[T-Data Triple] In this case $D = (a_1, a_2, a_3)^U$ and for every $i \in \{1, 2, 3\}$ it holds $\Gamma \vdash a_i : \mathit{Name}(U_i)$ and $V \preccurlyeq U$. So, from $W \preccurlyeq V$, we know that it holds $W \preccurlyeq U$. We conclude the proof, in this case, with an application of [T-Data Triple].

[T-Parallel Data] In this case $D = D_1 \mid D_2$ and for every $i \in \{1, 2\}$ it holds $\Gamma \vdash D_i : \mathit{Data}(V_i)$ and $V \preccurlyeq V_i$. So, from $W \preccurlyeq V$, we know that for every $i \in \{1, 2\}$

it holds $W \preccurlyeq V_i$. We apply the induction hypothesis and conclude the proof with an application of [T-Parallel Data]. □

The next lemma shows that if a data triple satisfies a pattern, which is well typed for W, then the privacy policy of the triple is less or equally restrictive than W and the triple is well typed for W. This result ensures that the data triples identified by a pattern are of the appropriate type and consequently used correctly in the continuation.

Lemma 4.4.5. *If* $(a, b, c)^V \models \chi$ *and* $\Gamma \vdash \chi : \mathit{Pattern}(W)$, *then* $W \preccurlyeq V$ *and* $\Gamma \vdash (a, b, c)^V : \mathit{Data}(W)$.

Proof. The proof is by induction on the derivation of $(a, b, c)^V \models \chi$.

[P-Triple] In this case $\chi = (a, b, c)^V$. We derive the proof from rule [T-Triple Pattern] and an application of rule [T-Data triple].

[P-OrL] In this case $\chi = \chi_1 \vee \chi_2$ and $(a, b, c)^V \models \chi_1$. From [T-Disjunction] we obtain $\Gamma \vdash \chi_1 : \mathit{Pattern}(W_1)$ and $W \preccurlyeq W_1$. By induction hypothesis we get $W_1 \preccurlyeq V$ and $\Gamma \vdash (a, b, c)^V : \mathit{Data}(W_1)$. So, by transitivity of $\preccurlyeq$, we get $W \preccurlyeq V$ and from Lemma 4.4.4 we conclude $\Gamma \vdash (a, b, c)^V : \mathit{Data}(W)$.

[P-OrR] Similar to the previous case.

[P-Exists] In this case $\chi = \exists x^U.\chi'$ and there exists d such that (H.1) $(a, b, c)^V \models \chi'\{d/x\}$ and $\mathcal{P}(d) = U$. From [T-Exists] we obtain (H.2) $\Gamma, x : \mathit{Name}(U) \vdash \chi' : \mathit{Pattern}(W)$. From $\mathcal{P}(d) = U$ and [T-Name] we get (H.3) $\Gamma \vdash d : \mathit{Name}(U)$. From (H.2),(H.3) and Lemma 4.4.3.1 we obtain (H.4) $\Gamma \vdash \chi'\{d/x\} : \mathit{Pattern}(W)$. Finally, we apply induction hypothesis on (H.1) and (H.4). □

The following lemmas show that data and names obtained by auxiliary functions, used for interaction with data, are well typed.

Lemma 4.4.6. *Let* $\Gamma \vdash D : \mathit{Data}(V)$. *If* $D' = \texttt{readable}(E, D)$, *then* $\Gamma \vdash D' : \mathit{Data}(V)$.

Proof. The proof is by induction on the structure of D.

$D = \emptyset_D$ In this case, from the definition of functions $\texttt{readable}(\cdot, \cdot)$ given in Figure 4.9, we conclude $D' = \emptyset_D$ and then from [T-Empty Data] we obtain $\Gamma \vdash D' : \mathit{Data}(V)$.

$D = (a, b, c)^W$ In this case we have two possible values for D'.

- If $D' = \emptyset_D$, then from [T-Empty Data] we obtain $\Gamma \vdash D' : \mathit{Data}(V)$.
- If $D' = (a, b, c)^W$, then $\Gamma \vdash D' : \mathit{Data}(V)$ holds trivially since $D' = D$.

$D = D_1 \mid D_2$ From [T-Parallel Data] we get that for every $i \in \{1, 2\}$ it holds $\Gamma \vdash D_i : \mathit{Data}(V_i)$ and $V \preccurlyeq V_i$. From the definition of function $\texttt{readable}(\cdot, \cdot)$ we get that $D' = D'_1 \mid D'_2$, where $D'_1 = \texttt{readable}(E, D_1)$ and $D'_2 = \texttt{readable}(E, D_2)$. From Lemma 4.4.4 and by induction hypothesis we get $\Gamma \vdash D'_1 : \mathit{Data}(V)$ and $\Gamma \vdash D'_2 : \mathit{Data}(V)$. We conclude the proof by an application of rule [T-Parallel Data]. □

Lemma 4.4.7. *Let* $\Gamma \vdash D : Data(V)$. *If* $D' = \texttt{delete}(\chi, D)$, *then* $\Gamma \vdash D' : Data(V)$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, from the definition of function $\texttt{delete}(\cdot,\cdot)$ given in Figure 4.9, we conclude $D' = \emptyset_D$ and then from [T-Empty Data] we obtain $\Gamma \vdash D' : Data(V)$.

$\boxed{D = (a, b, c)^W}$ In this case we have two possible values for D'.

- If $D' = \emptyset_D$, then from [T-Empty Data] we obtain $\Gamma \vdash D' : Data(V)$.
- If $D' = (a, b, c)^W$, then $\Gamma \vdash D' : Data(V)$ holds trivially since $D' = D$.

$\boxed{D = D_1 \mid D_2}$ In this case we derive the proof from [T-Parallel Data], Lemma 4.4.4, induction hypothesis and the definition of $\texttt{delete}(\cdot,\cdot)$. □

Lemma 4.4.8. *Let* $\Gamma \vdash \chi : Pattern(W)$. *If* $D' = \texttt{read}(\chi, D)$, *then* $\Gamma \vdash D' : Data(W)$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, from the definition of function $\texttt{read}(\cdot,\cdot)$ given in Figure 4.9, we conclude $D' = \emptyset_D$ and then from [T-Empty Data] we obtain $\Gamma \vdash D' : Data(W)$.

$\boxed{D = (a, b, c)^U}$ In this case we have two possible values for D'.

- If $D' = \emptyset_D$, then from [T-Empty Data] we obtain $\Gamma \vdash D' : Data(V)$.
- If $D' = (a, b, c)^U$, then, from the definition of function $\texttt{read}(\cdot,\cdot)$, we know $(a, b, c)^U \models \chi$. From Lemma 4.4.5 we conclude $\Gamma \vdash D' : Data(W)$.

$\boxed{D = D_1 \mid D_2}$ In this case we derive the proof from the definition of function $\texttt{read}(\cdot,\cdot)$, induction hypothesis and [T-Parallel Data]. □

Lemma 4.4.9. *Let* $S = \texttt{select}(\exists x^W.\chi, D)$. *If* $\{d/x\} \in S$, *then* $\Gamma \vdash d : Name(W)$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, from the definition of function $\texttt{select}(\cdot,\cdot)$ given in Figure 4.9, we conclude $S = \emptyset$. Therefore, $\{d/x\} \in S$ does not hold.

$\boxed{D = (a, b, c)^W}$ In this case we have two possible values for S.

- If $S = \emptyset$, then $\{d/x\} \in S$ does not hold.
- If there exists d such that $\{d/x\} \in S$, then, by the definition of function $\texttt{select}(\cdot,\cdot)$ we know that $d \in \{a, b, c\}$, $(a, b, c)^W \models \chi\{d/x\}$ and $\mathcal{P}(d) = W$. We conclude the proof with an application of rule [T-Name].

$\boxed{D = D_1 \mid D_2}$ In this case, by the definition of function $\texttt{select}(\cdot,\cdot)$ we obtain $S = \texttt{select}(\exists x^W.\chi, D_1 \mid D_2) = S_1 \cup S_2$, where $S_1 = \texttt{select}(\exists x^W.\chi, D_1)$ and $S_2 = \texttt{select}(\exists x^W.\chi, D_2)$. If there exists d such that $\{d/x\} \in S$, then $\{d/x\} \in S_1$ or $\{d/x\} \in S_2$. In any case we conclude $\Gamma \vdash d : \mathit{Name}(W)$ by induction hypothesis. □

Lemma 4.4.10. *Let* $\Gamma \vdash D : \mathit{Data}(V)$, $\Gamma \vdash \chi : \mathit{Pattern}(W)$ *and* $V \preccurlyeq W$. *If* $D' = \texttt{update}(\chi, D, W)$, *then*

1. $\Gamma \vdash D' : \mathit{Data}(V)$, *and*
2. *if* $D \models \texttt{ASK}(V)$, *then* $D' \models \texttt{ASK}(V)$.

Proof. The proof is by induction on the structure of D.

$\boxed{D = \emptyset_D}$ In this case, from the definition of $\texttt{update}(\cdot,\cdot,\cdot)$, we conclude $D' = \emptyset_D$ and then from [T-Empty Data] we obtain $\Gamma \vdash D' : \mathit{Data}(V)$. From Lemma 4.2.5, we know that it does not hold $D \models \texttt{ASK}(V)$.

$\boxed{D = (a,b,c)^U}$ In this case we have two possible values for D'.

- If $D' = (a,b,c)^U$ we derive the proof trivially, since $D' = D$.
- If $D' = (a,b,c)^W$, then, by the definition of $\texttt{update}(\cdot,\cdot,\cdot)$, we get $(a,b,c)^U \models \chi$. Then, from $\Gamma \vdash \chi : \mathit{Pattern}(W)$ and Lemma 4.4.5, we obtain $\Gamma \vdash (a,b,c)^U : \mathit{Data}(W)$. From that and rule [T-Data Triple] we get $\Gamma \vdash (a,b,c)^W : \mathit{Data}(W)$. From $V \preccurlyeq W$ and Lemma 4.4.4, we conclude $\Gamma \vdash D' : \mathit{Data}(V)$. From $D \models \texttt{ASK}(V)$ and [A-Triple] we get $D' \models \texttt{ASK}(V)$.

$\boxed{D = D_1 \mid D_2}$ In this case, for every $i \in \{1,2\}$ it holds $\Gamma \vdash D_i : \mathit{Data}(V_i)$ and $V \preccurlyeq V_i$. From Lemma 4.4.4 we derive that for every $i \in \{1,2\}$ it holds $\Gamma \vdash D_i : \mathit{Data}(V)$. We conclude $\Gamma \vdash D' : \mathit{Data}(V)$ from the definition of $\texttt{update}(\cdot,\cdot,\cdot)$, induction hypothesis and [T-Parallel Data]. From $D \models \texttt{ASK}(V)$, structural congruence rules and [A-Ask] we know that D_1 or D_2 satisfy $\texttt{ASK}(V)$. Then, by induction hypothesis we know that $\texttt{update}(\chi, D_1, W)$ or $\texttt{update}(\chi, D_2, W)$ satisfy $\texttt{ASK}(V)$. We conclude $D' \models \texttt{ASK}(V)$ from structural congruence rules and [A-Ask]. □

The following lemma proves that all interaction relations preserve well-typedness of involved processes and data, and that they preserve ownership of data. More precisely, the condition (3) ensures that the data which belong to a user stays fully readable to that user after the interaction happens. This includes the fact that the data might have changed during the interaction.

Lemma 4.4.11. *Let* $\emptyset \vdash a[D_a \parallel P_a] : \mathit{Network}$, $\emptyset \vdash b[D_b \parallel P_b] : \mathit{Network}$, $\mathcal{P}(a) = U$, $\mathcal{P}(b) = V$ *and* $(P_a, D_b) \rightsquigarrow_{a,b} (P'_a, D'_b)$. *Then*

(1) $\emptyset \vdash P'_a : \mathit{Process}(U)$, *and*

(2) $\emptyset \vdash D'_b : \mathit{Data}(V)$, *and*

(3) if $D'_b \neq \emptyset_D$, *then* $D'_b \models \texttt{ASK}(V)$.

Proof. If $D_a \neq \emptyset_D$, then from [T-User], we know $\emptyset \vdash a : \mathit{Name}(U)$, $\emptyset \vdash D_a : \mathit{Data}(U)$, $\emptyset \vdash P_a : \mathit{Process}(U)$, $D_a \models \texttt{ASK}(U)$. If $D_a = \emptyset_D$, then from [T-Blocked] and [T-Empty Data] we know $\emptyset \vdash a : \mathit{Name}(U)$, $\emptyset \vdash D_a : \mathit{Data}(U)$, $\emptyset \vdash P_a : \mathit{Process}(U)$. If $D_b \neq \emptyset_D$, then from [T-User], we know $\emptyset \vdash b : \mathit{Name}(V)$, $\emptyset \vdash D_b : \mathit{Data}(V)$, $\emptyset \vdash P_b : \mathit{Process}(V)$, $D_b \models \texttt{ASK}(V)$. If $D_b = \emptyset_D$, then from [T-Blocked] and [T-Empty Data] we know $\emptyset \vdash b : \mathit{Name}(V)$, $\emptyset \vdash D_b : \mathit{Data}(V)$, $\emptyset \vdash P_b : \mathit{Process}(V)$. Names a and b are not necessarily different.

The proof is by induction on the derivation of $(P_a, D_b) \rightsquigarrow_{a,b} (P'_a, D'_b)$.

[I-Read] In this case:

- $P_a = \texttt{READ}_b(\chi, X).P$;
- $P'_a = P\{D/X\}$ where $D = \texttt{read}(\chi, \texttt{readable}(D_a, D_b))$;
- $\emptyset \vdash \chi : \mathit{Pattern}(W)$;
- (H.1) $X : \mathit{Data}(W) \vdash P : \mathit{Process}(U)$;
- $D'_b = D_b$.

From Lemma 4.4.8 we obtain (H.2) $\emptyset \vdash D : \mathit{Data}(W)$. From (H.1), (H.2) and Lemma 4.4.3.3 we derive (1). (2) and (3) hold trivially, since $D'_b = D_b$.

[I-Write] In this case:

- $P_a = \texttt{WRITE}_b(D).P$;
- $P'_a = P$;
- $D_b \neq \emptyset_D$;
- $D'_b = D_b \mid D$;
- $\emptyset \vdash D : \mathit{Data}(W)$;
- (1) $\emptyset \vdash P : \mathit{Process}(U)$;
- $V \preccurlyeq W$.

We obtain (2) from Lemma 4.4.4 and [T-Parallel Data], and (3) from structural congruence rules for data and [A-Ask].

[I-Clear] In this case:

- $P_a = \texttt{CLEAR}_b$
- $P'_a = 0$;
- $D'_b = \emptyset_D$.

We obtain (1) from [T-Inaction] and (2) from [T-Empty Data]. (3) holds since $D_b = \emptyset_D$.

[I-Modify] In this case:

- $P_a = \texttt{MODIFY}_b(\chi, D).P$;
- $P'_a = P$;
- $D'_b = D' \mid D$ where $D' = \texttt{delete}(\chi, D_b)$;
- (1) $\emptyset \vdash P : \mathit{Process}(U)$;
- $U \preccurlyeq V$;
- $\emptyset \vdash D : \mathit{Data}(V)$;
- $D \models \texttt{ASK}(V)$.

From Lemma 4.4.7, we obtain $\emptyset \vdash D' : \mathit{Data}(V)$. So, from [T-Parallel Data] we obtain (2). From $D \models \texttt{ASK}(V)$, structural congurence rules and [A-Ask], we obtain (3).

[I-Select] In this case:

- $P_a = \texttt{SELECT}_b(\exists x^W.\chi, x).P$;
- $P'_a = \prod_{s \in S} Ps$ where $S = \texttt{select}(\exists x^W.\chi, \texttt{readable}(D_a, D_b))$;
- (H.1) $x : \mathit{Name}(W) \vdash P : \mathit{Process}(U)$;
- $D'_b = D_b$.

If $S \neq \emptyset$, then from (H.1), Lemma 4.4.9 and Lemma 4.4.3.2 we get that for every $s \in S$ it holds $\emptyset \vdash Ps : \mathit{Process}(U)$. We conclude (1) from [T-Parallel Process]. If $S = \emptyset$, then we conclude (1) from [T-Inaction]. (2) and (3) hold trivially, since $D'_b = D_b$.

[I-Update] In this case:

- $P_a = \texttt{UPDATE}_b(\chi, W).P$;
- $P'_a = P$;
- $D'_b = \texttt{update}(\chi, D_b, W)$;
- (1) $\emptyset \vdash P : \mathit{Process}(U)$;
- $\Gamma \vdash \chi : \mathit{Pattern}(W)$;
- $U \preccurlyeq V \preccurlyeq W$.

We obtain (2) and (3) from $\emptyset \vdash D_b : \mathit{Data}(V)$, $V \preccurlyeq W$ and by Lemma 4.4.10.

[I-Parallel] In this case:

- $P_a = P \mid Q$;
- $(P, D_b) \rightsquigarrow_{a,b} (P', D'_b)$;
- $P'_a = P' \mid Q$;
- $\Gamma \vdash P : \mathit{Process}(U)$;
- $\Gamma \vdash Q : \mathit{Process}(U)$.

We derive the proof by induction hypothesis and [T-Parallel Process].

[I-Choice] We derive the proof from [T-Choice].

[I-Structural] We derive the proof from Lemma 4.4.1, induction hypothesis, structural congruence rules and [A-Ask]. □

Subject reduction on networks is proved using Lemma 4.4.11. The lemma is applied twice, once for interaction of a user with its own data and once for interaction with the data of another user.

Theorem 4.4.12 (Subject reduction). *If* $\emptyset \vdash N : \mathit{Network}$ *and* $N \to N'$, *then* $\emptyset \vdash N' : \mathit{Network}$.

Proof. The proof is by induction on the derivation of $N \to N'$.

[R-User] In this case:

- $N = a[D_a \parallel P_a] \,[]\, b[D_b \parallel P_b]$;
- $N' = a[D_a \parallel P'_a] \,[]\, b[D'_b \parallel P_b]$;
- $a \neq b$;
- $(P_a, D_b) \rightsquigarrow_{a,b} (P'_a, D'_b)$;
- $\emptyset \vdash a : \mathit{Name}(U)$, $\emptyset \vdash D_a : \mathit{Data}(U)$, $\emptyset \vdash P_a : \mathit{Process}(U)$ and if $D_a \neq \emptyset_D$, then $D_a \models \texttt{ASK}(U)$;
- $\emptyset \vdash b : \mathit{Name}(V)$, $\emptyset \vdash D_b : \mathit{Data}(V)$, $\emptyset \vdash P_b : \mathit{Process}(V)$, and if $D_b \neq \emptyset_D$, then $D_b \models \texttt{ASK}(V)$.

From Lemma 4.4.11 we obtain $\emptyset \vdash P'_a : \mathit{Process}(U)$ and $\emptyset \vdash D'_b : \mathit{Data}(V)$ and if $D'_b \neq \emptyset_D$, then $D'_b \models \texttt{ASK}(V)$. If $D'_b \neq \emptyset_D$, we derive the proof, by rules [T-User] and [T-Parallel Network]. If $D'_b = \emptyset_D$, we derive the proof, by rules [T-Blocked] and [T-Parallel Network].

[R-Self] In this case:

- $N = a[D_a \parallel P_a]$;
- $N' = a[D'_a \parallel P'_a]$;
- $(P_a, D_a) \rightsquigarrow_{a,a} (P'_a, D'_a)$;

- $\emptyset \vdash a : \mathit{Name}(U)$, $\emptyset \vdash D_a : \mathit{Data}(U)$, $\emptyset \vdash P_a : \mathit{Process}(U)$, and if $D_a \neq \emptyset_D$, then $D_a \models \texttt{ASK}(U)$.

From Lemma 4.4.11 we obtain $\emptyset \vdash P'_a : \mathit{Process}(U)$ and $\emptyset \vdash D'_a : \mathit{Data}(U)$ and if $D'_a \neq \emptyset_D$, then $D'_a \models \texttt{ASK}(U)$. If $D'_a \neq \emptyset_D$, we derive the proof, by rule [T-User]. If $D'_a = \emptyset_D$, we derive the proof, by rule [T-Blocked].

[R-Struct] We derive the proof, in this case, from Lemma 4.4.1, induction hypothesis and structural congruence rules.

[R-Parallel] We derive the proof, in this case, by induction hypothesis and [T-Network]. □

Corollary 4.4.13. *If $\emptyset \vdash N : \mathit{Network}$ and $N \to^* N'$, then $\emptyset \vdash N' : \mathit{Network}$.*

We can now prove the soundness of the type system.

Theorem 4.4.14 (Type safety). *Let $\emptyset \vdash N : \mathit{Network}$. Then N is well behaved.*

Proof. Let $N \to^* a[D_a \parallel P_a] \,[]\, N'$ and $\mathcal{P}(a) = U$. By Corollary 4.4.13 and [T-Network Parallel], we know that (S.1) $\emptyset \vdash a[D_a \parallel P_a] : \mathit{Network}$ and (S.2) $\emptyset \vdash N' : \mathit{Network}$.

(1) Let $D_a \equiv (a_1, a_2, a_3)^V \mid E$. From (S.1), [T-User], [T-Parallel Data] and [T-Data Triple] we obtain $D_a \models \texttt{ASK}(U)$ and $U \preccurlyeq V$. From Theorem 4.3.2 we get $D \models \texttt{ASK}(U)$ implies $D \models \texttt{ASK}(V)$.

(2) Let $D_a = \emptyset_D$. We obtain $\texttt{readable}(D_a, D) = \emptyset_D$ from the definition of function $\texttt{readable}(\cdot,\cdot)$ From the definition of interaction relation it is obvious that $(\texttt{WRITE}_a(D), D_a)$ does not interact.

(3) Let $N' \equiv b[D_b \parallel P_b] \,[]\, N''$ and $\mathcal{P}(b) = V$. From (S.2) and the typing rules for networks, we know that (S.3) $\emptyset \vdash D_b : \mathit{Data}(V)$. From (S.1) and the typing rules for networks, we know that (S.4) $\emptyset \vdash P_a : \mathit{Process}(U)$.

So, if

- $P_a \equiv \texttt{MODIFY}_b(\chi, D).P \mid Q$, or
- $P_a \equiv \texttt{UPDATE}_b(\chi, V).P \mid Q$, or
- $P_a \equiv \texttt{CLEAR}_b \mid Q$,

from (S.4) and rule [T-Parallel Process] we obtain:

- $\emptyset \vdash \texttt{MODIFY}_b(\chi, D).P : \mathit{Process}(U)$,
- $\emptyset \vdash \texttt{UPDATE}_b(\chi, V).P : \mathit{Process}(U)$ and
- $\emptyset \vdash \texttt{CLEAR}_b : \mathit{Process}(U)$.

From rules [T-Change], [T-Update] and [T-Delete] we get that (S.5) $U \preccurlyeq V$. From (S.5) and Theorem 4.3.2 we get that $D \models \texttt{ASK}(U)$ implies $D \models \texttt{ASK}(V)$ for an arbitrary data D. In case $D \neq \emptyset_D$, from [T-User] we know $D_a \models \texttt{ASK}(U)$, so it also holds $D_a \models \texttt{ASK}(V)$.

□

4.5 Conclusions and related work

We have introduced a core language of processes that interact with data in RDF format, modelling a fragment of SPARQL and a higher order language that consumes the data. These processes together with data are enclosed under named users which are put in parallel, representing a network of users interacting with each other. We define the desired privacy properties of the network by defining well-behaved networks. What distinguishes this work from others analysing variety of security properties is the fact that it does not feature any additional means for privacy control beside those that are already present in the language, i.e. privacy policies are expressed as ask queries and policy satisfaction comes to query satisfaction with users profiles in RDF format. We have studied a type system guaranteeing some privacy properties by proving that well-typed networks are well behaved. The key of the type system simplicity lies in the introduced policy comparison relation and the chosen set of processes. As already mentioned, the calculus is equipped only with processes essential for the privacy study. With the extension of the process language we would get more complicated semantics and the type system without significant impact on the privacy properties which we have discussed.

This research contributes to the expanding trend of building the Web of Linked Data. To the best of our knowledge, this is the first typed calculus that tackles privacy protection for Linked Data. The proposed typed calculus provides a ground model for the development of type checkers for high level languages for Linked Data. When the Web of Linked Data becomes significantly spread out, the natural step forward will be the study of formal models of applications that consume the Web of Linked Data.

The Linked Data is currently present on the Web in many areas: media, publications, life sciences, geographic data and user generated content ([69, 15, 14, 12]). The presented research mostly focuses on privacy issues for user generated content such as posts, comments and images from social networks, discussion forums and blogs. There is a number of tools for users to share and manage their own sightings on a globally accessible database of linked data and for the content to be obtained by exporting data in RDF format. Here we name SIOC exporters [45] (Linked Data wrappers for blogging engines, content management systems and discussion forums), Zemanta (provides tools for the semiautomated enrichment of blog posts with data-level links pointing to DBpedia, Freebase, MusicBrainz, Semantic Crunch-Base, other blogs, etc.), OpenCalais (web service that extracts semantic metadata from text content, such as web pages), Graph API (provides Facebook social graph data in a semantically-enriched RDF format containing Linked Data), HyperTwitter (syntax and tool for embedding triple-like statements into Twitter microblogging messages) and Twarql (encodes information from microblog posts as Linked Open Data).

The syntax and the operational semantics of our calculus are inspired by RDF data format [36] and SPARQL and SPARQL Update query languages [122, 59]. Similar calculi appear in [31, 32] which provide of a syntax of scripting language for designing background processes that consume Linked Data , and in [42], which

studies provenance for Linked Data. We have introduced the user profiles and privacy policies as suggested in [127, 128, 134] where the authors propose the authentication of users based on their FOAF (language defining a dictionary of people-related terms that can be used in structured data) profiles and SPARQL ask queries as privacy preference checkers.

Type systems were successfully used to analyse a wide range of security properties (authenticity, security, safety, secrecy, privacy,...). We mention the following: in [1, 2] computational secrecy control was achieved for asynchronous π-calculus [104] by introducing types for public and private channels and proving computational soundness theorem; in [53] the type system for objective join-calculus guaranties that private labels are accessed only from the body of a class used to create the object; in [52] policy conformance in a distributed system is verified with a type system; in [64] authenticity and secrecy of well-typed protocols was proven for timed spi-calculus. The paper [31] provide static and dynamic typing with subtyping which identifies simple errors in the data and scripts and a minimal type inference algorithm. The most related type assignment systems are [41, 40], where the safety properties of data in XML format have been proved and [42] where types statically evaluate provenance driven access control.

In [41, 40], as well as in Chapter 3 of this thesis, type systems control security of data access in distributed networks with XML data. Matching function in [40] and Chapter 3 type checks identified data. Type system presented in the current chapter is able to statically check whether the data that will be identified will violate privacy or not. Such feature is due to simple data structure, refined syntax of patterns and syntax of processes which is more restrictive with respect to possible usages of the identified data.

In [42], the authors study provenance for Linked Data and introduce a typed calculus for modelling interaction between processes and Linked Data, tracing where the data has been published and who published it. The syntax of their calculus is similar to ours, in the sense that both calculi distinguish between processes and data and describe their interaction. In our calculus, processes run on behalf of data and both the data and the process are enclosed with a user name (like in named graphs), while in the other calculus linked data is open. Since the two calculi model orthogonal problems, triples are decorated with completely different annotations, while the tool for checking whether a data satisfy a query is the same. The semantics differs quite a lot since in our model it is essential that the operations are performed on the entire observed data. For example, think of updating a privacy policy: it must be updated on all triples satisfying some condition i.e. policy. Other related calculi do not have to guarantee this.

By means of a type system, we have been able to verify preservation of privacy properties taken from [127, 128], where the vocabulary for fine-grained privacy preference control and a tool for privacy preference management were defined, as well as several more general properties.

Chapter 5

Types for memory control

Overview This chapter is based on the paper "Exception Handling for Copyless Messaging" [89] and its early version [88]. In Section 5.1 we illustrate the problem we are addressing and informally sketch our solution in terms of types and a revised exception handling construct. In Section 5.2 we formally define the syntax and the semantics of a language of processes to model Sing# programs and we define well-behaved processes. Section 5.3 develops a type system for the language presented in Section 5.2 and Section 5.4 shows its soundness. Section 5.5 concludes the chapter with discussion and related work.

5.1 Copyless messaging and exceptions

Message passing is a flexible paradigm that allows autonomous entities to exchange information and to synchronize with each other. The term "message passing" seems to suggest a paradigm where messages *move* from one entity to another, although more often than not messages are in fact *copied* during communication. While this is inevitable in a distributed setting, the availability of a shared address space makes it possible to implement a *copyless* form of message passing, whereby only pointers to messages are exchanged.

The Singularity Operating System (Singularity OS) [84, 83] is a notable example of system that heavily relies on the copyless paradigm. In Singularity OS, processes have access to a shared region called the *exchange heap*, inter-process communication solely occurs by means of message passing over channels allocated on the exchange heap, and messages are themselves pointers to the exchange heap. As detailed by [84, 83, 50], it is not practical to automatically garbage collect objects on the exchange heap, which therefore must be explicitly managed by processes.

The copyless paradigm has obvious performance advantages over more conventional forms of message passing. At the same time, it fosters the proliferation of subtle programming errors arising from the explicit management of objects and the sharing of data. For this reason the designers of Singularity OS have equipped Sing#, the programming language used for the development of Singularity OS, with explicit constructs, types, and static analysis techniques to assist programmers in writing code that is free from a number of programming errors,

including: *memory faults*, namely the access to unallocated/deallocated objects in the heap; *memory leaks*, that is the accumulation of unreachable allocated objects in the heap; *communication errors*, which could cause the abnormal termination of processes and trigger the previous kinds of errors.

Some aspects of Sing# have already been formalized and studied by [44, 136, 146, 17]. In particular, in [17] it was shown that Sing# *channel contracts* can be conveniently represented as a variant of session types [78, 79], and that the information given by session types along with a linear type discipline can prevent memory leaks, memory faults, and communication errors. In [88, 89] we focus on *exceptions* and *exception handling*. The interest in our research stems from the observation that copyless messaging and exceptions are clearly at odds with each other: on the one hand, copyless messaging requires a very disciplined and controlled access to memory; on the other hand, exceptions are in general unpredictable and disrupt the normal control flow of programs. Consequently, and perhaps not surprisingly, these two aspects can be reconciled only with some native support from the runtime system. Here is a summary of the contributions of this research:

- we formalize a calculus of processes that communicate through the copyless paradigm and that can throw exceptions;
- we develop a type system for preventing the aforementioned errors even in the presence of exceptions, if suitable exception handlers are provided;
- we show how to take advantage of the invariants guaranteed by the type system in order to reduce the cost of exception handling.

5.1.1 Motivating example

To introduce the context in which we operate and the kind of problems we have to face, we take a look at a real fragment of Singularity OS. In the discussion that follows it is useful to keep in mind that Singularity channels consist of pairs of related *endpoints*, called the *peers* of the channel. Each endpoint is associated with an unbounded queue containing the messages sent to that endpoint from its peer. Communication is therefore asynchronous and send operations are nonblocking.

Figure 5.1 shows a Sing# function that computes the name for a newly allocated RAM disk. This function has been taken from `SourceControl/latest#base/Services/RamDisk/ClientManager/RamDiskClientManager.sg` in the Singularity OS source code available at `http://singularity.codeplex.com/`. Here, some identifiers are shortened in order to improve readability. The function has two output parameters, the computed disk name and the endpoint that links the disk to the `DirectoryService` (abbreviated `DS` in the code) which is part of the file system manager. The function begins by retrieving an endpoint `ns` for communicating with `DirectoryService` (line 3). Then the function repeatedly creates a new channel, represented as the peer endpoints `imp` and `exp` which are the output parameters of the `NewChannel` method (lines 6–8), computes a new disk name (line 9), and tries to register the chosen name along with `imp` to `DirectoryService`

```
1  void GetNextDiskPath(out string! diskName,
2                       out ServiceProviderContract.Exp! expService) {
3    DSContract.Imp:Ready ns = DS.NewClientEndpoint();
4    try {
5      while (true) {
6        ServiceProviderContract.Imp! imp;
7        ServiceProviderContract.Exp! exp;
8        ServiceProviderContract.NewChannel(out imp, out exp);
9        diskName = pathPrefix + nextDiskNumber.ToString();
10       ns.SendRegister(Bitter.FromString2(diskName), imp);
11       switch receive {
12         case ns.AckRegister():
13           nextDiskNumber++;
14           expService = exp;
15           return;
16         case ns.NakRegister(nakImp, error):
17           if (error == ErrorCode.AlreadyExists)
18             nextDiskNumber++;
19           else
20             throw new RamDiskErrorException(error);
21           delete exp;
22           delete nakImp;
23           break;
24       }
25     }
26   } finally {
27     delete ns;
28   }
29 }
```

Figure 5.1: Example of Sing# function.

```
contract DSContract {
  out message Success();
  in  message Register(char[]! in ExHeap path,
                       SPContract.Imp:Start! imp);
  out message AckRegister();
  out message NakRegister(SPContract.Imp:Start imp,
                          ErrorCode error);

  // ...more message types

  state Start : one { Success! → Ready; }

  state Ready : one {
    Register? → DoRegister;
    CreateDirectory? → ...
    // ...more transitions
  }

  state DoRegister : one {
    AckRegister! → Ready;
    NakRegister! → Ready;
  }
}
```

Figure 5.2: Example of Sing# contract.

through `ns` (line 10). The `switch receive` construct (lines 11–24) is used to receive messages and to dispatch control to various cases depending on the type of message that is received. Each `case` block specifies the endpoint from which a message is expected and the tag of the message. In this example, one of two kinds of messages are expected from the `ns` endpoint: either an `AckRegister` message (lines 12–15) or a `NakRegister` message (lines 16–23). In the first case the registration is successful (line 12), so the output parameter `expService` is properly initialized and the function terminates correctly (line 15). In the second case the registration is unsuccessful (line 16), hence a new registration is attempted if the error is recoverable (lines 17–18), otherwise an exception is thrown to abort the execution of the function (line 20). The main loop (lines 5–25) is protected within a `try` block with a `finally` clause that is executed regardless of whether the function terminates correctly or not. In the example, the clause deallocates the `ns` endpoint (line 27).

Sing# uses *channel contracts* to detect communication errors. Figure 5.2 shows (part of) the `DSContract` contract associated with endpoint `ns` in Figure 5.1. A contract is made of *message specifications* and of *states* connected by *transitions*. Each message specification begins with the keyword `message` and is followed by the *tag* of the message and the type of its arguments. In Figure 5.2, `DSContract`

defines the `Register` message with two arguments (a string and another endpoint) and the `AckRegister` message with no arguments. The `in` and `out` qualifiers specify the direction of messages from the point of view of the process exporting the contract. The state of the contract gives information about which messages can be sent/received at every given point in time. In `DSContract` we have a `Ready` state from which `Register`, `CreateDirectory`, and other (here omitted) messages can be received. After receiving a `Register` message, the contract moves to state `DoRegister`, from which one of the `AckRegister` or `NakRegister` messages can be sent, and then the contract goes back to the `Ready` state. In fact, each contract has two complementary views – called *exporting* and *importing* views – which are associated with the two peer endpoints of the channel. By convention, a contract declaration like that in Figure 5.2 specifies the exporting view of the contract: a provider of `DSContract` must adhere to its exporting view. In contrast, the function `GetNextDiskPath` in Figure 5.1 acts as a consumer of `DSContract`, therefore the function performs complementary actions by sending a `Register` message and then waiting for either an `AckRegister` or a `NakRegister` message. In the code, the importing and exporting views correspond to the types obtained by appending `.Imp` and `.Exp` suffixes to the name of the contract. For example, the declaration on line 3 specifies that `ns` is an endpoint having as type the importing view of `DSContract` in state `Ready`. After line 10, the type associated with `ns` changes to `DSContract.Imp:DoRegister` and then it goes back to `DSContract.Imp:Ready` after any of the receive operations on lines 12 and 16. Note that the changes in the state of the contract associated with `ns` (and therefore of the type of `ns`) are not explicit in the source code. They follow from the initial declaration that brings `ns` into scope (line 3) and from the way `ns` is used in the function. By keeping track of the contract state of `ns`, the compiler can statically check that the actions performed on `ns` (for sending and receiving messages) match corresponding co-actions (for receiving and sending) performed on its peer endpoint, which is in use by some other process in the system.

The code structure in Figure 5.1, involving channel allocation and deallocation, messaging, delegation (sending endpoints over other endpoints), and exception handling, is in fact typical throughout the whole Singularity OS and shows that these aspects are frequently mixed in non-trivial ways. We can identify two main problems caused by exceptions:

1. Since communication errors are prevented by the complementarity of actions performed by processes accessing peer endpoints, a jump in the control flow of one process, like that caused by an exception, may disrupt the alignment of the peers of a channel and compromise subsequent interactions.

2. When an exception is thrown, messages that have been sent but not yet received and other objects allocated since the beginning of a `try` may become unreachable and therefore turn into memory leaks.

Sing# has limited and not fully satisfactory mechanisms for dealing with these problems. Regarding the first, Sing# provides an `InState` method through which it is possible to query, at runtime, the actual state of an endpoint. This information can be used to attempt recovery from a possibly inconsistent state of the

endpoints. This mechanism implies an overhead for preserving and maintaining typing information at runtime and is unreliable as it depends on the programmer. The second problem seems to have been neglected. For example, the function in Figure 5.1 is prone to leak memory on line 20 in the case that the exception is thrown, since neither `exp` nor `nakImp` are properly deallocated (`imp` has been sent away in the call to `SendRegister` so it is not the current thread's responsibility to deallocate it). In this example it would suffice to move the `delete` instructions on lines 21 and 22 between lines 16 and 17 but, in general, it may be impossible to identify the exact point where an exception can be thrown and therefore when it is appropriate to deallocate resources. Note that it is unreasonable to assume that this clean-up code will be placed in the exception handler, if only because the handler may not be in the scope of the resources to be deallocated: in the example, `exp` and `nakImp` are not visible in the `finally` block so, by the time the exception has been thrown, it is too late to prevent the leak.

We put forward a solution that combines static analysis (inspired by existing works on exception handling for sessions by [25, 23, 24]) with a transaction-like, all-or-nothing semantics of `try` blocks. The basic idea is that a `try` block is either executed completely, and then its effects on the heap are committed and become permanent, or it is aborted by an exception. If this happens, *all* the processes involved in the transaction are notified of the exception, so that the types of the endpoints they are using can remain aligned, and the state of the heap is restored to that at the beginning of the `try` block. The solution relies on the following key ideas:

(1) Following [25, 23], we add explicit annotations to the types of endpoints used inside a transaction so that all processes involved in the transaction are aware of all the exceptions that can be thrown (possibly by a different process) during the transaction. In addition, these annotations make sure that the queues of endpoints used in a transaction are *empty* at the beginning of the transaction so that heap restoration solely amounts to removing messages from queues.

(2) Inside `try` blocks, we "seal" the type of any endpoint whose type is not properly annotated and we forbid processes to use endpoints with a sealed type. In this way, the type system can statically ensure that well-typed processes do not modify any portion of the heap outside the restorable one.

(3) We forbid the deallocation of endpoints inside `try` blocks, unless they have been allocated within the very same block. In this way, state restoration does not involve reallocations, which are difficult to implement correctly.

To prevent memory leaks, it is necessary to dynamically keep track of the memory allocated within a `try` block so that this memory can be properly reclaimed in case an exception is thrown. It is unsafe to deallocate an endpoint if its peer is not deallocated simultaneously: mechanism (1) guarantees that these deallocations are safe even if the type of these endpoints would not normally allow it, because transactions define a "closed scope" that includes, for each endpoint used in a transaction, also its peer. Starting with the next section we turn into the technical part, in which we make precise all of the concepts informally introduced so far.

P	$::=$		**Process**
		$\texttt{done}$	inaction
	$\mid$	$\texttt{open}(a, a).P$	open channel
	$\mid$	$\texttt{close}(u).P$	close endpoint
	$\mid$	$u!\mathtt{m}(u).P$	send
	$\mid$	$\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i$	receive
	$\mid$	$P \oplus P$	conditional
	$\mid$	$P \mid P$	parallel
	$\mid$	$\texttt{try}(U)\ \{\mathsf{e}_i : P_i\}_{i\in I} P$	initiate transaction
	$\mid$	$\texttt{throw}\ \mathsf{e}$	exception
	$\mid$	$\texttt{commit}(U).P$	commit transaction
	$\mid$	$X\langle\tilde{u}\rangle$	invocation
D	$::=$		**Definition**
		$X(\tilde{u}) \stackrel{\text{def}}{=} P$	rule

Figure 5.3: Syntax of processes and definitions.

5.2 Language

Notation We assume given an infinite set Pointers of *heap addresses* ranged over by a, b, ..., an infinite set Variables of *variables* ranged over by x, y, ..., and a set Exceptions of *exceptions* ranged over by e, ... We let *names* u, v, ... range over elements of Pointers $\cup$ Variables. We use A, B, ... to denote sets of pointers, $\mathcal{E}$, ... to denote sets of exceptions, U to denote sets of names, and $\tilde{u}$, $\tilde{v}$ to denote sequences of names (we will sometimes use $\tilde{u}$ to denote also the set of names in $\tilde{u}$). Process variables are ranged over by X, Y, ...

5.2.1 Syntax

The process language is essentially a variant of the π-calculus, except that names represent heap pointers instead of communicating channels. *Processes* are defined by the grammar in Figure 5.3. The term $\texttt{done}$ denotes the idle process that performs no action. The term $\texttt{open}(a, b).P$ denotes a process that allocates a new channel, represented as the two peer endpoints a and b, in the heap and continues as P. The term $u!\mathtt{m}(v).P$ denotes a process that sends the message $\mathtt{m}(v)$ on the endpoint u and then continues as P. A *message* is made of a *tag* $\mathtt{m}$ and an argument v. The term $\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i$ denotes a process that waits for a message from endpoint u. According to the tag $\mathtt{m}_i$ of the received message, the variable x_i is instantiated with the argument of the message in the continuation process P_i. We assume that the set I is always finite and non-empty. The terms $P \oplus Q$ and $P \mid Q$ are standard π-calculus processes. The term $\texttt{try}(U)\ \{\mathsf{e}_i : Q_i\}_{i\in I} P$ denotes a process willing to initiate a transaction involving the endpoints in U. The process P is the *body* of the transaction and is executed when the transaction is initiated, while the Q_i's are the *handlers* of the transaction which are activated if the transaction is aborted during the execution of the body by an exception e_i.

$$
\begin{aligned}
\mathsf{fn}(\texttt{done}) = \mathsf{fn}(\texttt{throw}) &= \emptyset \\
\mathsf{fn}(\texttt{open}(a,b).P) &= \mathsf{fn}(P) \setminus \{a,b\} \\
\mathsf{fn}(\texttt{close}(u).P) &= \{u\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(a!\mathtt{m}(b).P) &= \{a,b\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i) &= \{u\} \cup \bigcup_{i\in I}(\mathsf{fn}(P_i) \setminus \{x_i\}) \\
\mathsf{fn}(P \oplus Q) = \mathsf{fn}(P \mid Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}(\texttt{try}(U)\ \{\mathsf{e}_i : Q_i\}_{i\in I} P) &= U \cup \mathsf{fn}(P) \cup \textstyle\bigcup_{i\in I} \mathsf{fn}(Q_i) \\
\mathsf{fn}(\texttt{commit}(U).P) &= U \cup \mathsf{fn}(P) \\
\mathsf{fn}(X\langle\tilde{u}\rangle) &= \tilde{u}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{bn}(\texttt{done}) = \mathsf{bn}(\texttt{throw}) = \mathsf{bn}(X\langle\tilde{u}\rangle) &= \emptyset \\
\mathsf{bn}(\texttt{open}(a,b).P) &= \{a,b\} \cup \mathsf{bn}(P) \\
\mathsf{bn}(\texttt{close}(u).P) &= \mathsf{bn}(a!\mathtt{m}(b).P) = \mathsf{bn}(P) \\
\mathsf{bn}(\textstyle\sum_{i\in I} u?\mathtt{m}_i(x_i).P_i) &= \bigcup_{i\in I}(\{x_i\} \cup \mathsf{bn}(P_i)) \\
\mathsf{bn}(P \oplus Q) = \mathsf{bn}(P \mid Q) &= \mathsf{bn}(P) \cup \mathsf{bn}(Q) \\
\mathsf{bn}(\texttt{try}(U)\ \{\mathsf{e}_i : Q_i\}_{i\in I} P) &= \mathsf{bn}(P) \cup \textstyle\bigcup_{i\in I} \mathsf{bn}(Q_i) \\
\mathsf{bn}(\texttt{commit}(U).P) &= \mathsf{bn}(P)
\end{aligned}
$$

Figure 5.4: Free and bound names of processes.

The term $\texttt{throw}\ \mathsf{e}$ denotes the throwing of the exception e, whose effect is to abort the currently running transaction and to execute the appropriate handler. The term $\texttt{commit}(U).P$ denotes a process willing to terminate the currently running transaction (involving the endpoints in U). As soon as the transaction has ended, the process continues as P. The term $X\langle\tilde{u}\rangle$ denotes the invocation of the process associated with the process variable X. We assume that we work with a global environment of process definitions of the form

$$X(\tilde{u}) \stackrel{\text{def}}{=} P$$

defining these associations.

The binders of the language are $\texttt{open}(a,b).P$, which binds a and b in P, the input prefix $u?\mathtt{m}(x).P$, which binds x in P, and $X(\tilde{u}) \stackrel{\text{def}}{=} P$ which binds the names $\tilde{u}$ in P. The formal definitions of free and bound names of a process P, respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$, are standard and given in Figure 5.4. We identify processes modulo alpha renaming of bound names.

Syntactic conventions We adopt some standard conventions regarding the syntax of processes:

- we sometimes use an infix form for receive operations and write, for example $u?\mathtt{m}_1(x_1).P_1 + \cdots + u?\mathtt{m}_n(x_n).P_n$ instead of $\sum_{i=1..n} u?\mathtt{m}_i(x_i).P_i$;
- we sometimes use a prefix form for parallel compositions and write, for example, $\prod_{i=1..n} P_i$ instead of $P_1 \mid \cdots \mid P_n$;
- we identify $\prod_{i\in\emptyset} P_i$ with $\texttt{done}$;

$$
\begin{array}{l}
\texttt{GetNextDiskPath}(DS, ret) \stackrel{\text{def}}{=} \\
\quad DS?\texttt{NewClientEndpoint}(ns). \\
\quad \texttt{try}(ns)\ \{\texttt{RamDiskErrorException} : \texttt{Finally}\langle ns, DS, ret\rangle\} \\
\qquad \texttt{Loop}\langle ns, DS, ret\rangle \\
\\
\texttt{Loop}(ns, DS, ret) \stackrel{\text{def}}{=} \\
\quad \texttt{open}(imp, exp).ns!\texttt{Register}(imp). \\
\qquad ns?\texttt{AckRegister}().\texttt{commit}(ns). \\
\qquad\quad ret!\texttt{SetService}(exp).\texttt{Finally}\langle ns, DS, ret\rangle \\
\qquad + ns?\texttt{NakRegister}(nakImp). \\
\qquad\qquad \texttt{throw RamDiskErrorException} \\
\qquad\qquad \oplus\ \texttt{close}(exp).\texttt{close}(nakImp).\texttt{Loop}\langle ns, DS, ret\rangle \\
\\
\texttt{Finally}(ns, DS, ret) \stackrel{\text{def}}{=} \texttt{close}(ns).ret!\texttt{Result}(DS).\texttt{close}(ret)
\end{array}
$$

Figure 5.5: Encoding of the function in Figure 5.1.

– we omit message arguments when they are not used and we omit trailing occurrences of `done`, and write, for example, $a!\mathsf{a}().\texttt{close}(a)$ instead of $a!\mathsf{a}(c).\texttt{close}(a).\texttt{done}$

To ease the formalization, our process language supports a minimal set of critical features:

– we focus only on monadic messaging (messages have exactly one endpoint argument) and exception handling, disregarding other constructs and data types of Sing$^{\#}$;

– we assume that receive operations use the same endpoint in every branch, forbidding processes like $u?\mathsf{a}(x).P + v?\mathsf{b}(y).Q$ which are allowed by the `switch receive` construct in Sing$^{\#}$;

– we work with a purely prefix-based language without sequential composition, encoding `try-catch-finally` blocks in Sing$^{\#}$ with transaction bodies and handlers and `commit` processes within bodies;

– we encode `if-else` commands with the non-deterministic process $P \oplus Q$ omitting the condition that determines the chosen branch.

We claim that all the results presented hereafter can be suitably extended to overcome these restrictions.

Example 5.2.1. Figure 5.5 shows the encoding of the function in Figure 5.1 using the syntax of our process language. The structure of the process follows quite closely that of the function, except for some details which we explain here.

The loop on lines 5–25 is encoded as a recursive process `Loop` parameterized on its free names. The `finally` block on lines 26–28 is factored out as a named process `Finally`, since it must be executed regardless of whether the `try` block

is terminated successfully (line 15) or not (line 20). Consequently, `Finally` is invoked twice in the encoding.

The main difference between the function Figure 5.1 and its encoding concerns parameter passing, which is encoded using explicit communication on the *ret* endpoint. In particular, the initialization of `expService` with `exp` on line 14 corresponds to the output operation $ret!\texttt{SetService}(exp)$ in Figure 5.5.

Note that in Figure 5.1 the function uses a global name `DS` for accessing a system service. In order to obtain a closed term, in the encoding we explicitly mention a parameter *DS* of the `GetNextDiskPath` process which represents `DS`. Because our type system relies on the linear access to resources, invoking a parametric process such as `GetNextDiskPath` means transferring the ownership of the parameters to the process. To preserve linearity (of *DS* in this case), the `Finally` process sends *DS* back on *ret* before *ret* is closed (more involved examples of function modeling and ownership transfer are described in detail by [17]). △

5.2.2 Operational semantics

In order to describe the operational semantics of processes, we need to represent the *heap* where channels are allocated and through which messages are exchanged. Indeed, channels are accessed through the pointers to their endpoints and message arguments are themselves pointers to heap objects. Intuitively, a heap μ is a finite map from pointers a to endpoint structures $[b, \mathfrak{Q}]$, where b is the *peer endpoint* of a and $\mathfrak{Q}$ is the queue of messages waiting to be received from a. In the model, we represent heaps and message queues as terms generated by the grammar in Figure 5.6. The term $\emptyset$ denotes the empty heap, in which no endpoints are allocated. The term $a \mapsto [b, \mathfrak{Q}]$ denotes an endpoint allocated at a pointing to the endpoint structure $[b, \mathfrak{Q}]$. The term μ, μ' denotes the composition of the heaps μ and μ'. We write $\mathsf{dom}(\mu)$ for the *domain* of the heap μ, that is the set of pointers for which there is an allocated endpoint structure. The heap composition μ, μ' is well defined provided that $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') = \emptyset$ (there cannot be two endpoint structures allocated at the same address). In the following, we identify queues assuming associativity of composition and the laws $\varepsilon :: \mathfrak{Q} = \mathfrak{Q} :: \varepsilon = \mathfrak{Q}$ and we identify heaps assuming associativity and commutativity of composition and the law $\emptyset, \mu = \mu$. We write $a \mapsto [b, \mathfrak{Q}] \in \mu$ to indicate that the endpoint structure $[b, \mathfrak{Q}]$ is allocated at location a in μ.

Message queues, ranged over by $\mathfrak{Q}$, are also represented as terms: ε denotes the empty queue, $\mathtt{m}(c)$ is a queue made of an `m` message with argument c, and $\mathfrak{Q} :: \mathfrak{Q}'$ is the queue composition of $\mathfrak{Q}$ and $\mathfrak{Q}'$. We identify queues modulo associativity of :: and we assume that ε is neutral for ::.

Before defining the operational semantics of processes we formalize two notions. The first one is that of peer endpoints:

Definition 5.2.2 (peer endpoints). *We say that a and b are* peer endpoints *in μ, written $a \overset{\mu}{\leftrightarrow} b$, if $a \neq b$ and $a \mapsto [b, \mathfrak{Q}] \in \mu$ and $b \mapsto [a, \mathfrak{Q}'] \in \mu$.*

Note that $\overset{\mu}{\leftrightarrow}$ is a symmetric relation.

The notion of "closed scope" that we mentioned in Section 5.1.1 is formalized as a predicate on sets of pointers:

μ	$::=$		**Heap**
		$\emptyset$	empty heap
	$\mid$	$a \mapsto [a, \mathfrak{Q}]$	endpoint structure
	$\mid$	μ, μ	heap composition
$\mathfrak{Q}$	$::=$		**Queue**
		ε	empty queue
	$\mid$	$\mathtt{m}(a)$	message
	$\mid$	$\mathfrak{Q} :: \mathfrak{Q}$	queue composition
P	$::=$		**Runtime process**
		$\cdots$	as in Figure 5.3
	$\mid$	$\langle A, A, \{\mathsf{e}_i : P_i\}_{i \in I} P \rangle$	running transaction

Figure 5.6: Syntax of heaps, queues, and runtime processes.

$$\mathsf{fn}(\langle A, B, \{\mathsf{e}_i : Q_i\}_{i \in I} P \rangle) = A \cup B \cup \mathsf{fn}(P) \bigcup_{i \in I} \mathsf{fn}(Q_i)$$
$$\mathsf{bn}(\langle A, B, \{\mathsf{e}_i : Q_i\}_{i \in I} P \rangle) = \mathsf{bn}(P) \bigcup_{i \in I} \mathsf{bn}(Q_i)$$

Figure 5.7: Free and bound names of the running transaction.

Definition 5.2.3 (balanced set of pointers). *We say that* $A \subseteq \mathsf{dom}(\mu)$ *is* balanced *in* μ*, written* μ-$\mathsf{balanced}(A)$*, if, for every* $a \in A$*,* $a \overset{\mu}{\leftrightarrow} b$ *implies* $b \in A$*.*

Informally, A is balanced in μ if for every a in A, the peer of a is also in A provided that it is still allocated in μ. Since a message sent over a ends up in the queue of its peer, this means that any communication occurring on one of the endpoints in A remains within the scope identified by A.

In the operational semantics of processes, we need to distinguish between a transaction that has not started yet (and which is represented using the `try` construct of Figure 5.3), and a *running transaction*. This need arises for two reasons: First, a running transaction generally involves more than one process, each with its own set of handlers. Therefore, it is technically convenient to devise an explicit construct that defines the *scope* of the transaction. Second, it is necessary to keep track of the part of the heap that has been allocated since the initiation of the transaction. Figure 5.6 extends the syntax of processes with the term $\langle A, B, \{\mathsf{e}_i : P_i\}_{i \in I} P \rangle$ where A is the set of endpoints involved in the transaction, B is the set of endpoints that have been allocated since the transaction has started, P is the (residual) body of the transaction, and the P_i's represent the handlers of the transaction. In general, P and the P_i's will be parallel compositions of the bodies and the handlers of the processes that have cooperatively initiated the transaction. Free and bound names of the running transaction are defined in Figure 5.7

The operational semantics of processes is defined in terms of a structural congruence over processes (identifying structurally equivalent processes) and a reduction relation. Structural congruence is the least relation $\equiv$ including alpha

Structural congruence

[S-Par Idle] $P \mid \texttt{done} \equiv P$

[S-Par Comm] $P \mid Q \equiv Q \mid P$

[S-Par Assoc] $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

Reduction relation

[R-Open] $\mu \fatsemi \texttt{open}(a, b).P \to \mu, a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon] \fatsemi P$

[R-Parallel] $$\frac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi P \mid Q \to \mu' \fatsemi P' \mid Q}$$

[R-Close] $\mu, a \mapsto [b, \mathfrak{Q}] \fatsemi \texttt{close}(a).P \to \mu \fatsemi P$

[R-Choice] $$\frac{i \in \{1, 2\}}{\mu \fatsemi P_1 \oplus P_2 \to \mu \fatsemi P_i}$$

[R-Send] $\mu, a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}'] \fatsemi a!\mathtt{m}(c).P \to \mu, a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \mathtt{m}(c)] \fatsemi P$

[R-Receive] $$\frac{k \in I}{\mu, a \mapsto [b, \mathtt{m}_k(c) :: \mathfrak{Q}] \fatsemi \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i \to \mu, a \mapsto [b, \mathfrak{Q}] \fatsemi P_k\{c/x_k\}}$$

[R-Invoke] $$\frac{X(\tilde{u}) \stackrel{\text{def}}{=} P}{\mu \fatsemi X\langle \tilde{a} \rangle \to \mu \fatsemi P\{\tilde{a}/\tilde{u}\}}$$

[R-Struct] $$\frac{P \equiv P' \qquad \mu \fatsemi P' \to \mu' \fatsemi Q' \qquad Q' \equiv Q}{\mu \fatsemi P \to \mu' \fatsemi Q}$$

Figure 5.8: Operational semantics of processes.

conversion and the laws in Figure 5.8, stating that parallel composition is commutative, associative, and has `done` as neutral element. As process interaction mostly occurs through the heap, the reduction relation describes the evolution of *configurations* $\mu \fatsemi P$ rather than of processes alone, so that

$$\mu \fatsemi P \to \mu' \fatsemi P'$$

denotes the fact that process P evolves to P' and, in doing so, it changes the heap from μ to μ'.

Reduction is the smallest relation between configurations defined by the rules in Figures 5.8 and 5.9.

We explain the rules in the following paragraphs. Rule [R-Open] describes the creation of a new channel, which causes the allocation of two new endpoint structures in the heap. The endpoints are initialized with empty queues and are allocated at fresh locations, for otherwise the resulting heap would be ill formed. Since we have assumed that Pointers is infinite, it is always possible to alpha rename a and b to fresh pointers using structural congruence, so that an $\texttt{open}(a, b).P$ is always able to reduce.

[R-Start Transaction]

$$\frac{\mu\text{-balanced}(\bigcup_{i\in I} A_i)}{\mu \fatsemi \prod_{i\in I} \texttt{try}(A_i)\ \{\mathsf{e}_j : Q_{ij}\}_{j\in J} P_i \to \mu \fatsemi \langle \bigcup_{i\in I} A_i, \emptyset, \{\mathsf{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i\rangle}$$

[R-End Transaction]

$$\mu \fatsemi \langle A, B, \{\mathsf{e}_j : Q_j\}_{j\in J} \prod_{i\in I} \texttt{commit}(A_i).P_i\rangle \to \mu \fatsemi \prod_{i\in I} P_i$$

[R-Run Transaction]

$$\frac{\mu \fatsemi P \to \mu' \fatsemi P'}{\mu \fatsemi \langle A, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle \to \mu' \fatsemi \langle A, \mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu')), \{\mathsf{e}_j : Q_j\}_{j\in J} P'\rangle}$$

[R-Catch Exception]

$$\frac{k \in J}{\mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2 \fatsemi \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e}_k \mid P\rangle \to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I} \fatsemi Q_k}$$

[R-Propagate Exception]

$$\frac{\forall j \in J : \mathsf{e}_j \neq \mathsf{e}}{\mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2 \fatsemi \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e} \mid P\rangle \to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I} \fatsemi \texttt{throw}\ \mathsf{e}}$$

Figure 5.9: Operational semantics of transactions.

Rule [R-Close] describes the closing of an endpoint, which deallocates its structure from the heap and discards its queue. Note that both endpoints of a channel are created simultaneously by [R-Open], but each is closed independently by [R-Close] (this is the same semantics as the one of $\mathsf{Sing}^{\#}$).

Rule [R-Choice] states that a process $P \oplus Q$ nondeterministically reduces to either P or Q.

Rule [R-Send] describes the sending of a message $\mathtt{m}(c)$ on the endpoint a. The message is enqueued at the right end of the queue associated with the peer endpoint b of a. Note that, for this rule to be applicable, it is necessary for both endpoints of a channel to still be allocated.

Rule [R-Receive] describes the receiving of a message from endpoint a. In particular, the message at the left end of the queue associated with a is removed from the queue, its tag $\mathtt{m}_k$ is used to select one branch of the process, and its argument c instantiates the corresponding variable x_k.

Rule [R-Parallel] describes the independent evolution of parallel processes. Note how the heap is treated globally even if there is only one subprocess to reduce.

Rule [R-Invoke] describes process invocations simply as the replacement of a process variable with the process it is associated with, modulo the substitution of its parameters. In this rule and in [R-Receive], $P\{\tilde{a}/\tilde{b}\}$ denotes the capture-avoiding substitution of $\tilde{a}$ in place of $\tilde{b}$ in P.

Rule [R-Start Transaction] describes the initiation of a transaction by a number of processes. The transaction is identified by a set of endpoints $\bigcup_{i\in I} A_i$

which are distributed among the processes. In order for the transaction to start, this set of endpoints must be balanced, so that for every endpoint in the set its peer is also in the set. The rule is nondeterministic, in the sense that there can be multiple combinations of processes that can initiate a transaction. We leave the choice of a particular strategy (for example, requiring $\bigcup_{i\in I} A_i$ to be non-empty, minimal, and μ-$\mathsf{balanced}$) to the implementation. The residual process is the tuple

$$\langle \bigcup_{i\in I} A_i, \emptyset, \{\mathsf{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i \rangle$$

combining the bodies and the handlers of the processes involved in the transaction. The second component is $\emptyset$ indicating that at this stage no new endpoints have been allocated yet within the transaction. Note that the combined processes must be able to handle the same set $\{\mathsf{e}_j\}_{j\in J}$ of exceptions for the reduction to occur. Even if this seems to require a runtime check, the fact that all processes involved in a transaction are able to handle the same set of exceptions will be ensured by the type system (see rule [T-Try] in Figure 5.13).

Rule [R-End Transaction] reduces a running transaction to its continuation when its body has terminated. The handlers are discarded. The sets A_i play no role in the operational semantics and are used for typing purposes only. In fact, we will see that the type system enforces the invariant that these sets coincide with the ones decorating the `try` blocks corresponding to the `commit` processes and, in particular, $A = \bigcup_{i\in I} A_i$.

Rule [R-Run Transaction] allows the reduction of a transaction according to the reductions of its body. The rule keeps track of the memory changes occurred during the reduction of the body of the transaction by updating the second component of the transaction to $\mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu'))$, where

$$\mathsf{track}(B, A_0, A_1) \stackrel{\text{def}}{=} (B \cup (A_1 \setminus A_0)) \setminus (A_0 \setminus A_1)$$

In practice, the pointers to objects allocated during the reduction are added to B, while the pointers to objects deallocated during the reduction are removed from B.

Rule [R-Catch Exception] describes the abnormal termination of a running transaction when an exception is thrown and the transaction provides a handler for it. In this case, the queues of all the endpoints involved in the transactions are emptied, the memory allocated within the transaction is reclaimed, and the appropriate handler is run. In a similar way, rule [R-Propagate Exception] abnormally terminates running transactions when there is no suitable handler for the thrown exception. Also in this case the queues of the endpoints involved in the transactions are emptied and the memory allocated within the transaction is reclaimed, but the exception is propagated (technically, *re-thrown*) at the outer level.

We write $\mu \fatsemi P \rightarrow$ if $\mu \fatsemi P \rightarrow \mu' \fatsemi P'$ for some μ' and P' and $\mu \fatsemi P \not\rightarrow$ if not $\mu \fatsemi P \rightarrow$; we write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$.

5.2.3 Well-behaved processes

We conclude this section providing a characterization of *well-behaved processes*, those that are free from memory leaks, memory faults, and communication errors.

$$\text{[ST-Inactive]}\quad \mu \fatsemi \texttt{done} \downarrow$$

$$\text{[ST-Input]}\quad \mu, a \mapsto [b, \varepsilon] \fatsemi \textstyle\sum_{i \in I} a?\mathtt{m}_i(x_i).P_i \downarrow$$

$$\text{[ST-Commit]}\quad \mu \fatsemi \texttt{commit}(A).P \downarrow$$

$$\text{[ST-Try]}\quad \frac{\neg \mu\text{-}\mathsf{balanced}(A)}{\mu \fatsemi \texttt{try}(A)\ \{\mathsf{e}_j : Q_j\}_{j \in J} P \downarrow}$$

$$\text{[ST-Parallel]}\quad \frac{\mu \fatsemi P \downarrow \qquad \mu \fatsemi Q \downarrow}{\mu \fatsemi P \mid Q \downarrow}$$

$$\text{[ST-Running Transaction]}\quad \frac{\mu \fatsemi P \downarrow \qquad P \not\equiv \prod_{i \in I} \texttt{commit}(A_i).P_i}{\mu \fatsemi \langle A, B, \{\mathsf{e}_j : Q_j\}_{j \in J} P \rangle \downarrow}$$

Figure 5.10: Stuck configurations.

A *memory leak* occurs when no pointer to an allocated region of the heap is retained by any process. In this case, the allocated region has no owner, it occupies space, but it is no longer accessible. A *memory fault* occurs when a pointer is accessed and the endpoint it points to is not (or no longer) allocated. A *communication error* occurs when some process receives a message of unexpected type. To formalize well-behaved processes, we need to define the reachability of a heap object with respect to a set of *root* pointers. Intuitively, a process P may directly reach any object located at some pointer in the set $\mathsf{fn}(P)$ (we can think of the pointers in $\mathsf{fn}(P)$ as of the local variables of the process stored on its stack); from these pointers, the process may reach other heap objects by reading messages from the endpoints it can reach, and so forth.

Definition 5.2.4 (reachable pointers). *We say that c is* reachable *from a in μ, notation $c \prec_\mu a$, if $a \mapsto [b, \mathfrak{Q} :: \mathtt{m}(c) :: \mathfrak{Q}'] \in \mu$. We write $\preccurlyeq_\mu$ for the reflexive, transitive closure of $\prec_\mu$ and we define μ-$\mathsf{reach}(A) = \{c \in \mathsf{Pointers} \mid \exists a \in A : c \preccurlyeq_\mu a\}$.*

The last auxiliary notion we need provides a syntactic characterization of those configurations that cannot reduce but that do not represent any of the errors described above.

Definition 5.2.5 (stuck configuration). *We say that the configuration $\mu \fatsemi P$ is* stuck *if the judgment $\mu \fatsemi P \downarrow$ is inductively derivable by the rules in Figure 5.10.*

Rules [ST-Inactive] and [ST-Parallel] are obvious, while rules [ST-Try] and [ST-Commit] state that transaction initiations and termination are stuck, if taken in isolation. In the former case, the set of involved endpoints must not be balanced, for otherwise the transaction could initiate. Rule [ST-Running Transaction] states that a running transaction is stuck if its body is stuck and different from a combination of processes willing to terminate the transaction, for otherwise the transaction could terminate. Note also that no exception can have been thrown within the body of a stuck running transaction, for the stuckness predicate is undefined for `throw` e processes. Finally, rule [ST-Input] states that a process waiting for a message from endpoint a is stuck only if the endpoint a is allocated and its queue is empty. Then, a configuration whose processes are all waiting for a message corresponds to a genuine deadlock. From these rules we deduce that a

process willing to send a message on a is never stuck, and so is a process willing to receive a message from a if the queue associated with a is not empty.

Definition 5.2.6 (well-behaved process). *We say that* P *is* well behaved *if* $\emptyset \fatsemi P \to^* \mu \fatsemi Q$ *implies:*

(1) $\mathsf{dom}(\mu) \subseteq \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$*;*

(2) $Q \equiv Q_1 \mid Q_2$ *and* $\mu \fatsemi Q_1 \not\to$ *imply* $\mu \fatsemi Q_1 \downarrow$*.*

Less formally, a process P is well behaved if every residual Q of P is such that Q can reach every pointer in the heap and every subprocess Q_1 of Q that does not reduce is stuck (recall that the definition of $\mu \fatsemi Q_1 \downarrow$ captures also the possibility that Q_1 is in deadlock). Here are a few examples of ill-behaved processes to illustrate the sort of errors we want to spot with our type system:

- The process $\mathtt{open}(a, b).\mathtt{done}$ violates condition (1), since it leaks endpoints a and b.
- The process $\mathtt{open}(a, b).(\mathtt{close}(a).\mathtt{close}(a) \mid \mathtt{close}(b))$ tries to deallocate the same endpoint a twice. This is an example of fault.
- The process $\mathtt{open}(a, b).(a!\mathtt{a}().\mathtt{close}(a) | b?\mathtt{b}().\mathtt{close}(b))$ violates condition (2) since it reduces to a parallel composition of subprocesses where one has sent an $\mathtt{a}$ message, but the other one was expecting a $\mathtt{b}$ message.
- The process

$$\begin{array}{l}\mathtt{open}(a, b).\mathtt{try}(\emptyset)\ \{\mathsf{e} : \mathtt{done}\} \\ \qquad\qquad \mathtt{throw}\ \mathsf{e} \oplus \mathtt{commit}(\emptyset).\mathtt{close}(a).\mathtt{close}(b)\end{array}$$

  may leak a and b if the exception is thrown.

Observe that, in item (1) of Definition 5.2.6, the domain of μ is only required to be *included in* (instead of being *equal to*) the set of pointers reachable from the free names of Q. In particular, it may be the case that Q contains references to unallocated objects, and yet it never attempts to use them. This formulation of leak-freedom, which is slightly more general than the one used by [88] where equality between the two sets was required, is necessary because the type system that we are about to define allows subtyping, which was not considered in [88].

Notice that our notion of leak-freedom does not require a process to eventually deallocate the objects it owns, but only to guarantee the reachability of all the objects it owns. For example, the process $\mathtt{open}(a, b).X\langle a, b\rangle$ where $X(a, b) \stackrel{\text{def}}{=} X\langle a, b\rangle$, maintains the reachability of a and b without ever using them. This process is well behaved according to Definition 5.2.6 and is also well typed according to the type system that will be developed in Section 5.3.

t	$::=$		**Type**
		T	endpoint type
	$\mid$	$[t]$	sealed type
T	$::=$		**Endpoint type**
		$\texttt{end}$	termination
	$\mid$	α	type variable
	$\mid$	$\{!\mathtt{m}_i(T_i).T_i\}_{i\in I}$	internal choice
	$\mid$	$\{?\mathtt{m}_i(T_i).T_i\}_{i\in I}$	external choice
	$\mid$	$\{\mathtt{e}_i : T_i\}_{i\in I}[\![T$	initiate transaction
	$\mid$	$]\!]T$	commit transaction
	$\mid$	$\texttt{rec}\ \alpha : r.T$	recursive type
	$\mid$	$\{\mathtt{e}_i : T_i\}_{i\in I}T$	running transaction

Figure 5.11: Syntax of types and endpoint types.

5.3 Type system

We now develop a type system that enforces well-behavedness of processes: in Section 5.3.1 we introduce the syntax of the type language; in Section 5.3.2 we define a notion of *type weight* which is used for discriminating between safe and unsafe communications; Section 5.3.3 is devoted to extending classical subtyping for session types by [57] so as to take transactions and exceptions into account; Sections 5.3.4, 5.3.5, and 5.3.6 define the actual typing rules.

5.3.1 Syntax of types

We assume given an infinite set of *type variables* ranged over by α; we use t, s, ... to range over types, and T, S, ... to range over endpoint types. The syntax of types and endpoint types is defined in Figure 5.11. An endpoint type describes the behavior of a process with respect to a particular endpoint: the process may send messages over the endpoint, receive messages from the endpoint, deallocate the endpoint, initiate and terminate transactions involving the endpoint. The endpoint type $\texttt{end}$ denotes an endpoint that can only be deallocated. An internal choice $\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint on which a process may send any message $\mathtt{m}_i$ for $i \in I$. The message has an argument of type S_i and, depending on the tag $\mathtt{m}_i$, the endpoint can be used thereafter according to T_i. In a dual manner, an external choice $\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ denotes an endpoint from which a process must be ready to receive any message $\mathtt{m}_i$ for $i \in I$ and, depending on the tag $\mathtt{m}_i$ of the received message, the endpoint is to be used according to T_i. In endpoint types $\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and $\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ we assume that $I \neq \emptyset$ and $\mathtt{m}_i = \mathtt{m}_j$ implies $i = j$ for every $i, j \in I$. That is, the tag $\mathtt{m}_i$ of the message that is sent or received identifies a unique continuation T_i. The endpoint type $\{\mathtt{e}_i : S_i\}_{i\in I}[\![T$ denotes an endpoint on which it is possible to initiate a transaction. The type T specifies how the endpoint is used within the body of the transaction, whereas each type S_i specifies how the endpoint is used if the transaction is aborted by the exception $\mathtt{e}_i$. The

$$\text{[WF-End]}\quad \Theta \vdash \mathtt{end} : 0 \qquad \text{[WF-Var]}\quad \Theta, \alpha : r \vdash \alpha : r \qquad \text{[WF-Rec]}\quad \frac{\Theta, \alpha : r \vdash T : r}{\Theta \vdash \mathtt{rec}\ \alpha : r.T : r}$$

$$\text{[WF-Prefix]}\quad \frac{\dagger \in \{?, !\} \qquad \Theta \vdash S_i : 0\ ^{(i \in I)} \qquad \Theta \vdash T_i : r\ ^{(i \in I)}}{\Theta \vdash \{\dagger \mathtt{m}_i(S_i).T_i\}_{i \in I} : r} \qquad \text{[WF-Commit]}\quad \frac{\Theta \vdash T : r}{\Theta \vdash]\!]T : r + 1}$$

$$\text{[WF-Initiate]}\quad \frac{\Theta \vdash S_i : r\ ^{(i \in I)} \qquad \Theta \vdash T : r + 1}{\Theta \vdash \{\mathtt{e}_i : S_i\}_{i \in I}[\![T : r} \qquad \text{[WF-Run]}\quad \frac{\Theta \vdash S_i : r\ ^{(i \in I)} \qquad \Theta \vdash T : r + 1}{\Theta \vdash \{\mathtt{e}_i : S_i\}_{i \in I} T : r}$$

Figure 5.12: Rank of endpoint types.

endpoint type $]\!]T$ denotes the termination of the transaction in which an endpoint with this type is involved. As soon as the transaction is properly terminated, the endpoint can be subsequently used according to T. Terms α and $\mathtt{rec}\ \alpha : r.T$ can be used to specify recursive behaviors, as usual. The annotation r associated with α represents the rank of α, which will be explained shortly. Finally, the endpoint type $\{\mathtt{e}_i : S_i\}_{i \in I} T$ is analogous to $\{\mathtt{e}_i : S_i\}_{i \in I}[\![T$, except that it specifies the type of an endpoint involved in a transaction which has already been initiated, but has not terminated yet. In fact, this type is needed for technical reasons only, and will be used in conjunction with running transaction processes $\langle A, B, \{\mathtt{e}_i : P_i\}_{i \in I} P \rangle$. In no case the programmer is supposed to deal with endpoint types of this form.

Clearly, not every endpoint type written according to the syntax in Figure 5.11 makes sense. For example, it is possible to write terms such as $\{\mathtt{e} : \mathtt{end}\}[\![\mathtt{end}$ where a transaction is initiated but not terminated or terms where recursions do not respect the intended nesting of transactions, like in $\mathtt{rec}\ \alpha.\{\mathtt{e} : \mathtt{end}\}[\![\alpha$ or in $\{\mathtt{e} : \mathtt{end}\}[\![\mathtt{rec}\ \alpha.]\!]\alpha$. As far as our analysis is concerned, the syntax does not even prevent `end` subterms from occurring within transactions, which as we have argued in Section 5.1.1 is undesirable since endpoints involved in transactions should not be closed. For all these reasons we define a subset of *well-formed* endpoint types based on a notion of *rank*. Intuitively, the rank of a term T gives the number of transactions within which T may occur, with the proviso that `end` must have rank 0.

In general, we say that the endpoint type T is well formed and has rank r in Θ if $\Theta \vdash T : r$ is inductively derivable by the axioms and rules in Figure 5.12, where Θ ranges over ranking contexts associating ranks to type variables. Then, a derivation of $\emptyset \vdash T : 0$ means that T is a closed endpoint type where transaction initiations and terminations are balanced. Rules [WF-Initiate], [WF-Run], and [WF-Commit] count the number of nested transactions. Rule [WF-Prefix] requires all branches of a choice to have the same rank, while rules [WF-Rec] and [WF-Var] deal with recursive types in a standard way, by respectively augmenting and accessing the ranking context. In the following we will omit Θ from judgments $\Theta \vdash T : r$ if Θ is empty.

As welcome side effects of well formedness, note that:

- message types have rank 0 (rule [WF-Prefix]). Thus, well-typed processes will not be able to send/receive endpoints involved in pending transactions;
- `end` cannot occur inside transactions (rule [WF-End]). Thus, well-typed processes will not be able to close endpoints involved in pending transactions.

The rank annotation r in recursive terms $\texttt{rec}\ \alpha : r.T$ guarantees that every well-formed endpoint type has a uniquely determined rank. Without this annotation a term like $\texttt{rec}\ \alpha.!\texttt{m}(\texttt{end}).\alpha$ could be given any rank. The following proposition guarantees that the rank of well-formed endpoint types is unaffected by folding/unfolding of recursions:

Proposition 5.3.1. *If* $\vdash \texttt{rec}\ \alpha : r.T : r$, *then* $\vdash T\{\texttt{rec}\ \alpha : r.T/\alpha\} : r$.

Proof. A simple induction on the derivation of $\Theta, \alpha : r \vdash T : r$. □

In what follows, we will assume that all endpoint types are closed and well formed and we will usually omit the rank annotation from recursive terms with the assumption that they can be properly annotated so that they are well formed; we will also write $\mathsf{rank}(T)$ for the rank of T. We will identify endpoint types modulo alpha renaming of bound type variables (the only binder being `rec`) and folding/unfolding of recursions knowing that this does not change their rank (Proposition 5.3.1). In particular, we have $\texttt{rec}\ \alpha.T = T\{\texttt{rec}\ \alpha.T/\alpha\}$. Finally, we will sometimes use an infix notation for internal and external choices and write $!\texttt{m}_1(S_1).T_1 \oplus \cdots \oplus !\texttt{m}_n(S_n).T_n$ instead of $\{!\texttt{m}_i(S_i).T_i\}_{i\in\{1,\ldots,n\}}$ and $?\texttt{m}_1(S_1).T_1 + \cdots + ?\texttt{m}_n(S_n).T_n$ instead of $\{?\texttt{m}_i(S_i).T_i\}_{i\in\{1,\ldots,n\}}$.

Types are possibly sealed endpoint types of the form

$$[\cdots[T]\cdots]$$

for some arbitrary (possibly zero) number of seals $[\cdots]$. Seals protect the endpoints not involved in a transaction: they are applied when the transaction is initiated (the `try` primitive is executed) and are stripped off when the transaction terminates (the `commit` primitive is executed). The type system prevents endpoints with a sealed type from being used, since any change to them would not be undoable in case the currently running transaction is aborted.

Example 5.3.2. According to the process definitions in Figure 5.5, the endpoint *ns* is involved in the transaction around the `Loop` process, it is used for sending a `Register` message and then for receiving either an `AckRegister` or a `NakRegister` message. The same endpoint is then closed regardless of whether the transaction completes successfully or not. We can describe the overall behavior of `Loop`, `GetNextDiskPath` and `Finally` on *ns* with the following endpoint type:

$$T_{ns} = \{\texttt{RamDiskErrorException} : \texttt{end}\}[\![\texttt{rec}\ \alpha.!\texttt{Register}(T_{imp}).\\ (?\texttt{AckRegister}().]\!]\texttt{end} + ?\texttt{NakRegister}(T_{imp}).\alpha)$$

where T_{imp} is the endpoint type associated with the *imp* and *nakImp* endpoints.

The endpoint *ret* is not used within the transaction, but its usage differs depending on whether or not the exception is thrown:

$$T_{ret} = !\texttt{Result}(T_{DS}).\texttt{end} \oplus !\texttt{SetService}(T_{exp}).!\texttt{Result}(T_{DS}).\texttt{end}$$

If no exception is thrown, *ret* is used for sending a `SetRegister` message followed by a `Result` one; if an exception is thrown, only the `Result` message is sent. The above type T_{ret} takes into account both possibilities. △

In order to avoid communication errors, we associate peer endpoints with endpoint types describing complementary actions: if a process sends a message of some kind on one endpoint, another process must be able to receive a message of that kind from the peer endpoint; if one process initiates a transaction involving one endpoint, the other process will do so as well on the peer endpoint; if one process has finished using an endpoint, the process owning the peer endpoint has finished too. We formalize this complementarity of actions by defining a function that, given an endpoint type, computes its dual:

Definition 5.3.3 (duality). Duality *is the function* $\overline{\cdot}$ *on endpoint types defined coinductively by the equations:*

$$\begin{array}{rcl}
\overline{\texttt{end}} & = & \texttt{end} \\
\overline{\{?\texttt{m}_i(S_i).T_i\}_{i\in I}} & = & \{!\texttt{m}_i(S_i).\overline{T_i}\}_{i\in I} \\
\overline{\{!\texttt{m}_i(S_i).T_i\}_{i\in I}} & = & \{?\texttt{m}_i(S_i).\overline{T_i}\}_{i\in I} \\
\overline{\{\texttt{e}_i : S_i\}_{i\in I}[\![T}} & = & \{\texttt{e}_i : \overline{S_i}\}_{i\in I}[\![\overline{T} \\
\overline{]\![T} & = &]\![\overline{T} \\
\overline{\{\texttt{e}_i : S_i\}_{i\in I}T} & = & \{\texttt{e}_i : \overline{S_i}\}_{i\in I}\overline{T}
\end{array}$$

Roughly speaking, the dual of an endpoint type T is obtained from T by swapping internal and external choices. For example, the dual of the endpoint type T_{ret} defined in Example 5.3.2 is

$$\overline{T_{ret}} = ?\texttt{Result}(T_{DS}).\texttt{end} + ?\texttt{SetService}(T_{exp}).?\texttt{Result}(T_{DS}).\texttt{end}$$

Note that the dual $\overline{T}$ of T cannot be defined by a simple induction on the structure of T according to this intuition because the type of message arguments is *un*affected by duality. In particular we have

$$\begin{array}{rcl}
\overline{\texttt{rec}\ \alpha.?\texttt{m}(\alpha).\texttt{end}} & = & \overline{?\texttt{m}(\texttt{rec}\ \alpha.?\texttt{m}(\alpha).\texttt{end}).\texttt{end}} \\
& = & !\texttt{m}(\texttt{rec}\ \alpha.?\texttt{m}(\alpha).\texttt{end}).\texttt{end} \\
& \neq & \texttt{rec}\ \alpha.!\texttt{m}(\alpha).\texttt{end}\,.
\end{array}$$

In [17] one can find an equivalent inductive definition of duality.

We list here two important properties of duality, namely that it is an involution and it preserves ranks:

Proposition 5.3.4. *The following properties hold:*

1. $\overline{\overline{T}} = T$;
2. $\mathsf{rank}(\overline{T}) = \mathsf{rank}(T)$.

Proof. Item (1) is an easy consequence of the definition of duality (Definition 5.3.3). Item (2) follows from the fact that the rank is only affected by the nesting of transaction types in T and internal/external choices are treated in the same way by rule [WF-Prefix]. □

5.3.2 Type weight

In [17] it was observed that the delegation of endpoints having some particular types can generate memory leaks even if the delegating process appears to behave correctly with respect to the type of the endpoints it uses. For example, the process

$$P \stackrel{\text{def}}{=} \texttt{open}(a, b).a!\mathtt{m}(b).\texttt{close}(a) \tag{5.1}$$

uses a and b according to the endpoint types

$$T = !\mathtt{m}(S).\texttt{end} \qquad \text{and} \qquad S = \texttt{rec}\ \alpha : 0.?\mathtt{m}(\alpha).\texttt{end} \tag{5.2}$$

respectively. Note that $\overline{T} = S$, therefore the complementarity of actions performed on the peer endpoints a and b is guaranteed. Now, the process P sends endpoint b over endpoint a. According to T, the process is indeed entitled to send an $\mathtt{m}$ message with argument of type S on a and b has precisely that type. After the output operation, the process no longer owns endpoint b and endpoint a is deallocated. Despite its apparent correctness, P generates a leak, as shown by the reduction:

$$\begin{array}{rcl} \emptyset \fatsemi P & \to & a \mapsto [b, \varepsilon], b \mapsto [a, \varepsilon] \fatsemi a!\mathtt{m}(b).\texttt{close}(a) \\ & \to & a \mapsto [b, \varepsilon], b \mapsto [a, \mathtt{m}(b)] \fatsemi \texttt{close}(a) \\ & \to & b \mapsto [a, \mathtt{m}(b)] \fatsemi \texttt{done} \end{array}$$

In the final configuration we have $\mu\text{-}\mathsf{reach}(\mathsf{fn}(\texttt{done})) = \emptyset$ while $\mathsf{dom}(\mu) = \{b\}$. In particular, the endpoint b is no longer reachable and therefore this configuration violates condition (1) of Definition 5.2.6. A closer look at the heap in the reduction above reveals that the problem lies in the cycle involving b resulting from the send operation $a!\mathtt{m}(b)$: it is as if the $b \mapsto [a, \mathtt{m}(b)]$ region of the heap needs not be owned by any process because "it owns itself". Fortunately, it is possible to detect the situations in which these cycles may be generated by looking at the structure of the types of the endpoints that are sent as messages. More specifically, for each endpoint type we compute a value in the set $\mathbb{N} \cup \{\infty\}$, which we call *weight*, that estimates the length of any chain of pointers originating from the queue of the endpoints it denotes. A weight equal to ∞ means that this length can be infinite, in the sense that cycles such as the one shown above may be generated. Then, the type system makes sure that only endpoints having a finite-weight type can be sent as messages, and this has been shown, in [17] to be enough for preventing these kinds of memory leaks.

We proceed by recalling here the definition of weight from [17], adapted to our context where we deal also with transaction types:

Definition 5.3.5 (weight). *We say that $\mathcal{W}$ is a* coinductive weight bound *if $(T, n) \in \mathcal{W}$ implies either:*

- $T = \texttt{end}$ *or* $T = \{\mathsf{e}_i : S_i\}_{i \in I}[\![T'$ *or* $T =]\!]T'$ *or* $T = \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}$, *or*

- $T = \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ *and* $n > 0$ *and* $(S_i, n-1) \in \mathcal{W}$ *and* $(T_i, n) \in \mathcal{W}$ *for every* $i \in I$*, or*
- $T = \{\mathtt{e}_i : S_i\}_{i\in I}T'$ *and* $(T', n) \in \mathcal{W}$*.*

We write $T :: n$ *if* $(T, n) \in \mathcal{W}$ *for some coinductive weight bound* $\mathcal{W}$*. The* weight *of an endpoint type* T*, denoted by* $\|T\|$*, is defined by*

$$\|T\| = \min\{n \in \mathbb{N} \mid T :: n\}$$

where we let $\min \emptyset = \infty$*. When comparing weights we extend the usual total orders* $<$ *and* $\leq$ *over natural numbers so that* $n < \infty$ *for every* $n \in \mathbb{N}$ *and* $\infty \leq \infty$*.*

The weight of T is defined as the least of its weight bounds, or ∞ if there is no such weight bound. For example we have $\|\mathtt{end}\| = \|\{!\mathtt{m}_i(S_i).T_i\}_{i\in I}\| = 0$. Indeed, the queues of endpoints with type `end` and those in a send state are empty and therefore the chains of pointers originating from them have zero length. The same happens for endpoints whose type is $\{\mathtt{e}_i : S_i\}_{i\in I}[\![T$ and $]\!]T$, since we will enforce the invariant that when a transaction is initiated or successfully terminated, the endpoints involved in it have empty queues. Endpoint types in a receive state have a strictly positive weight. For instance we have $\|?\mathtt{m}(\mathtt{end}).\mathtt{end}\| = 1$ and $\|?\mathtt{m}(?\mathtt{m}(\mathtt{end}).\mathtt{end}).\mathtt{end}\| = 2$. Indeed, the queue of an endpoint with type $?\mathtt{m}(\mathtt{end}).\mathtt{end}$ may contain another endpoint with an empty queue. Therefore, the chain of pointers originating from the endpoint with type $?\mathtt{m}(\mathtt{end}).\mathtt{end}$ has at most length 1. If we go back to the endpoint types in (5.2) that we used to motivate this discussion, we have $\|T\| = 0$ and $\|S\| = \infty$, from which we deduce that endpoints with type S, like b in (5.1), are not safe to be used as messages.

5.3.3 Subtyping

The last notion we need before proceeding with the definition of the type system is a *subtyping relation* for endpoint types. Because of the close relationship between endpoint types and session types, the subtyping relation for endpoint types turns out to be a variant of that for session types [57]. However, the peculiar nature of exceptions has interesting consequences. The original subtyping relation for session types is based on the fundamental duality between input and output actions. In particular, it establishes that subtyping is *covariant* for external choices (inputs) and *contravariant* for internal ones (outputs). For example,

$$T = !\mathtt{a}(S_1).T_1 \oplus !\mathtt{b}(S_2).T_2 \leqslant !\mathtt{a}(S_1).T_1 = S$$

is a valid subtyping relation between T and S. The underlying intuition is based on the usual principle of safe substitution of an endpoint of type S with another endpoint of type T. If a (well-typed) process is using an endpoint c of type S, then it can only send an `a` message on c. So, replacing the endpoint c with another one of type T, which allows both `a` and `b` messages to be sent, does not compromise communication safety. In a dual manner,

$$T' = ?\mathtt{a}(S_1).T_1 \leqslant ?\mathtt{a}(S_1).T_1 + ?\mathtt{b}(S_2).T_2 = S'$$

is a valid subtyping relation between T' and S'. In this case, a (well-typed) process using an endpoint of type S' must be capable of handling (at least) `a` and `b` messages received from the endpoint. Replacing that endpoint with another one of type T' is safe because from the latter one only `a` messages can be received.

The covariance and contravariance properties of subtyping with respect to input and output operations follow from the duality of endpoint types associated with peer endpoints: when a process is entitled to send a message on an endpoint, the process using its peer must be ready to receive it, and vice versa. By contrast, during a transaction, exceptions can be thrown on both peers of a channel. As a consequence, the two transaction types

$$\{\mathtt{e}_1 : S_1, \mathtt{e}_2 : S_2\}[\![T \qquad \text{and} \qquad \{\mathtt{e}_1 : S_1\}[\![T$$

cannot be related. Indeed, if we had $\{\mathtt{e}_1 : S_1, \mathtt{e}_2 : S_2\}[\![T \leqslant \{\mathtt{e}_1 : S_1\}[\![T$, then the process using the endpoint a with type $\{\mathtt{e}_1 : S_1\}[\![T$ might not be prepared to handle the exception $\mathtt{e}_2$ thrown by the process using the peer b of a. Similarly, the process using the peer endpoint b might be unable to handle the exception $\mathtt{e}_2$ thrown on a if we had the opposite relation. In the end, because of the bi-directional nature of exceptions thrown during a transaction, subtyping must be *invariant* for transaction types.

We now proceed to define subtyping formally, extending it to possibly sealed endpoint types in the natural way:

Definition 5.3.6 (subtyping). Subtyping *is the largest relation* $\leqslant$ *such that* $T \leqslant s$ *implies either:*

- $t = [t']$ *and* $s = [s']$ *and* $t' \leqslant s'$*, or*
- $t = s = \mathtt{end}$*, or*
- $t = \{?\mathtt{m}_i(T_i).T'_i\}_{i \in I}$ *and* $s = \{?\mathtt{m}_i(S_i).S'_i\}_{i \in I \cup J}$ *and* $T_i \leqslant S_i$ *and* $T'_i \leqslant S'_i$ *for every* $i \in I$*, or*
- $t = \{!\mathtt{m}_i(T_i).T'_i\}_{i \in I \cup J}$ *and* $s = \{!\mathtt{m}_i(S_i).S'_i\}_{i \in I}$ *and* $S_i \leqslant T_i$ *and* $T'_i \leqslant S'_i$ *for every* $i \in I$*, or*
- *either* ($t = \{\mathtt{e}_i : T_i\}_{i \in I}[\![T$ *and* $s = \{\mathtt{e}_i : S_i\}_{i \in I}[\![S$*) or* ($t = \{\mathtt{e}_i : T_i\}_{i \in I}T$ *and* $s = \{\mathtt{e}_i : S_i\}_{i \in I}S$*) and* $T_i \leqslant S_i$ *for every* $i \in I$ *and* $T \leqslant S$*, or*
- $t =]\![T$ *and* $s =]\![S$ *and* $T \leqslant S$*.*

According to the definition of $\leqslant$, the covariance and contravariance properties for external and internal choices informally introduced earlier are extended to message argument types, in the usual manner. Observe that subtyping is always covariant with respect to continuations. It is easy to show that $\leqslant$ is a pre-order that is contravariant with respect to duality:

Proposition 5.3.7. *The following properties hold:*

1. $\leqslant$ *is reflexive and transitive;*
2. $T \leqslant S$ *if and only if* $\overline{S} \leqslant \overline{T}$*.*

Proof. See [57]. Transaction types do not pose additional issues. □

In Section 5.3.2 we have introduced a notion of weight that will be used in the type system for discriminating between safe and unsafe messages. Since the weight is computed on the (static) type of endpoints and subtyping allows for the substitution of endpoints with related but possibly different types, one important question arises whether subtyping and type weight are coherent with each other. This is indeed the case:

Proposition 5.3.8. *$T \leqslant S$ implies $\|T\| \leq \|S\|$.*

Proof. It is easy to see that $\mathcal{W} = \{(T, n) \mid \exists S : T \leqslant S \wedge S :: n\}$ is a coinductive weight bound. In particular, when T is an internal choice we have $T :: 0$ regardless of the number of branches in T. □

5.3.4 Typing processes

We can now proceed to defining a type system for processes.

A *type environment* is a finite map $\Gamma = \{u_i : T_i\}_{i \in I}$ from names to types. We write $\mathsf{dom}(\Gamma)$ for the domain of Γ, namely the set $\{u_i\}_{i \in I}$; we write Γ, Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$; finally, we write $\Gamma \vdash u : T$ if $\Gamma(u) = T$. An *exception environment* $\tilde{\mathcal{E}}$ is a finite sequence $\mathcal{E}_1 \cdots \mathcal{E}_n$ of sets of exceptions. We write $\mathsf{e} \in \tilde{\mathcal{E}}$ if $\mathsf{e} \in \mathcal{E}_k$ for some $k \in \{1, \ldots, n\}$. We say that a type t is *local*, written $\mathsf{local}(t)$, if t is not sealed and has a null rank, namely $t = T$ for some T such that $\mathsf{rank}(T) = 0$. Intuitively, a local type denotes an endpoint that can be modified (its type is not sealed) and is not involved in any transaction. We extend the notion of local types to type environments so that $\mathsf{local}(\Gamma)$ holds if every type in the codomain of Γ is local.

The typing rules for processes are inductively defined in Figure 5.13. Judgments have the form

$$\tilde{\mathcal{E}}; \Gamma \vdash P$$

denoting that process P is well typed in the exception environment $\tilde{\mathcal{E}}$ and type environment Γ. In particular, P can only throw exceptions that occur in $\tilde{\mathcal{E}}$. The type system makes use of a global process environment Σ associating process variables X with pairs $(\tilde{t}, \tilde{\mathcal{E}})$ containing the type of the parameters of X as well as the exception environment $\tilde{\mathcal{E}}$ in which X is supposed to be invoked. It is understood that the process environment Σ contains associations for all the global definitions D and that the judgment $\Sigma \vdash D$ defined by

$$\frac{\Sigma(X) = (\tilde{t}, \tilde{\mathcal{E}}) \qquad \tilde{\mathcal{E}}; \tilde{u} : \tilde{t} \vdash P}{\Sigma \vdash X(\tilde{u}) \stackrel{\text{def}}{=} P}$$

holds. In particular, *all* of the free names of P must occur in its binding variable $X(\tilde{u})$.

We describe the typing rules for processes in the following paragraphs. Rule [T-Inaction] states that the idle process is well typed only in the empty type environment. This is a standard rule for linear type systems implying, in our case, that the terminated process has no leaks.

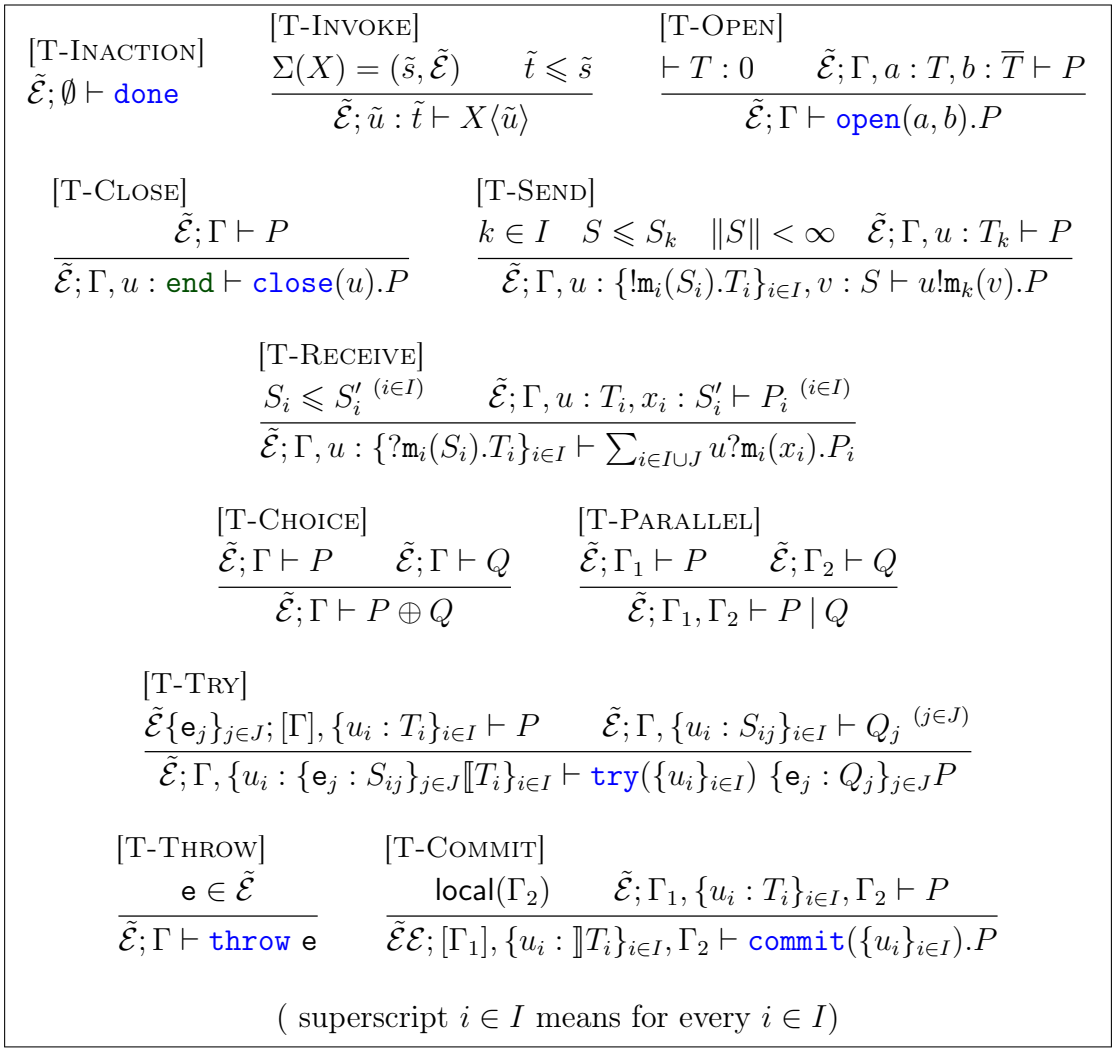

Figure 5.13: Typing rules for processes.

Rule [T-Invoke] declares that a process invocation $X\langle\tilde{u}\rangle$ is well typed provided that the number and type of actual parameters $\tilde{u}$ match the number and type of formal parameters in $\Sigma(X)$ and that the process is invoked in the correct exception environment. In this rule we write $\tilde{t} \leqslant \tilde{s}$ for the pointwise extension of $\leqslant$ to sequences of types.

Rule [T-Open] deals with the creation of a new channel, which is visible in the continuation process as two peer endpoints typed by dual endpoint types. The premise $\vdash T : 0$ means that newly created endpoints have no pending transactions on them.

Rule [T-Close] states that a process $\texttt{close}(u).P$ is well typed provided that u corresponds to an endpoint with type `end`, on which no further interaction is possible, and P is well typed in the remaining type environment.

Rule [T-Send] states that a process $u!\mathtt{m}(v).P$ is well typed if u is associated with an endpoint type T that permits the output of `m` messages. The type S of the argument v must be unsealed, finite-weight, and has to be a subtype of the expected type in the endpoint type. Finally, the continuation P must be well typed in a type environment where the endpoint u is typed according to the continuation T_k of T and the endpoint v is no longer visible. This models the fact that the ownership of v is transferred to the process that receives the message.

Rule [T-Receive] deals with inputs: a process waiting for a message from an endpoint $u : \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ is well typed if it can deal with all of the messages $\mathtt{m}_i$. The continuation processes may use the endpoint u according to the endpoint type T_i and can access the message argument x_i of of some supertype S_i' of S_i.

Rules [T-Choice] and [T-Parallel] are standard. In the latter, the type environment is split into two disjoint environments to type the processes being composed.

We now turn our attention to the constructs dealing with transactions and exceptions.

Rule [T-Try] deals with transaction initiations. All the endpoints in the decoration U must have a type allowing them to be involved in a transaction, while the types of other names are sealed so that P is prevented from using them until the transaction is terminated. Seals are not applied in the type environment for the handlers since they execute only if and when the transaction is aborted and therefore act outside of the transaction. Note that the admitted exceptions are augmented in P but not in Q.

Rule [T-Throw] states that the process `throw e` is well typed in any type environment, provided that it occurs within a transaction (the exception being thrown must be among the ones occurring in the exception environment). For this reason, the violation of linearity for the assumptions in the type environment is only apparent, as control will be transferred at runtime to some appropriate exception handler.

Rule [T-Commit] is almost the dual of rule [T-Try] and deals with transaction termination. Again, the endpoints in the decoration U must have a matching type in the context indicating the end of the transaction. Names with a sealed type must have been inherited from the context surrounding the transaction being terminated, so a seal is stripped off them in the continuation P. Names with a local

type must have been created within the transaction being terminated, and can be used in the continuation as well. Note that the rightmost set $\mathcal{E}$ in the exception environment is stripped off when type checking P, since P executes after the transaction has terminated hence outside of the scope where the exceptions in $\mathcal{E}$ can be thrown.

Observe that the type system requires the endpoints specified in a `commit` process to be exactly the same as the ones in the corresponding `try`. This is a consequence of the properties of well-formed endpoint types: endpoints involved in a transaction have a type with a strictly positive rank (see [WF-Initiate] in Figure 5.12) meaning that they cannot be closed (because `end` has null rank) and they cannot be sent as messages (again because [WF-Prefix] requires messages to have a type with null rank). For the same reason they cannot be qualified as local, because local endpoints have a type with a null rank. Therefore, the set $\{u_i\}_{i\in I}$ associated with a given `try` process will be exactly the same set associated with the corresponding `commit` process.

Example 5.3.9. Using the types defined in Example 5.3.2, the reader can verify that the bodies of the process definitions in Figure 5.5 for `GetNextDiskPath`, `Loop`, and `Finally` are respectively well typed according to the type environments

$$\begin{array}{lcl} \Gamma_1 & = & DS : ?\texttt{NewClientEndpoint}(T_{ns}).T_{DS},\, ret : T_{ret} \\ \Gamma_2 & = & ns : T'_{ns},\, DS : T_{DS},\, ret : T_{ret} \\ \Gamma_3 & = & ns : \texttt{end},\, DS : T_{DS},\, ret : T_{ret} \end{array}$$

where

$$T'_{ns} = !\texttt{Register}(T_{imp}).(?\texttt{AckRegister}().]\!]\texttt{end} + ?\texttt{NakRegister}(T_{imp}).T_{ns})$$

is an appropriate residual of the unfolding of T_{ns}. Note the role played by subtyping in this example: *ret* is used according to the type $!\texttt{Result}(T_{DS}).\texttt{end}$ in `Finally` and according to the type $!\texttt{SetService}(T_{exp}).!\texttt{Result}(T_{DS}).\texttt{end}$ in `Loop`. Since T_{ret} is a subtype of both these types, *ret* can be passed to `Finally` and `Loop` thanks to the subtyping relation in [T-Invoke]. △

5.3.5 Typing the heap

The typing rules in Figure 5.13 are not sufficient for proving the soundness of the type system, because they are solely concerned with the static syntax of processes. At runtime, we must take into account running transaction processes (see Figures 5.6 and 5.9) as well as the heap. Indeed, since inter-process communication relies on heap-allocated structures, several properties of well-behaved processes depend on properties of the heap saying that its content is consistent with a given type environment. In this section and in the following one we develop a type system for the runtime components of our process language. We remark that the programmer is solely concerned with the typing rules for static processes presented in Section 5.3.4, while the technical material presented hereafter, which builds on and extends the previous one, is only required for proving that the type system is sound.

Just as we have type checked a process P against a type environment that associates types with the names occurring in P, we also need to check that the heap is consistent with respect to the same environment. This leads to a notion of well-typed heap that we develop in this section. More precisely, well-typedness of a heap μ is checked with respect to a pair $\Gamma_0;\Gamma$ of type environments: the context Γ_0,Γ must provide type information for *all* the allocated structures in μ (that is, $\mathsf{dom}(\Gamma_0,\Gamma) = \mathsf{dom}(\mu)$); the splitting $\Gamma_0;\Gamma$ distinguishes the pointers in $\mathsf{dom}(\Gamma)$ from the pointers in $\mathsf{dom}(\Gamma_0)$ so that Γ contains the *roots* of μ, namely the pointers that are not referenced from any endpoint structure in the heap, while Γ_0 contains pointers that are referenced from some endpoint structure.

Among the properties that must be enforced is the complementarity between the endpoint types associated with peer endpoints. This notion of complementarity does not coincide with duality because the communication model is asynchronous: since messages can accumulate in the queue of an endpoint before they are received, the types of peer endpoints can be misaligned. The two peers are guaranteed to have dual types only when their queues are both empty. In general, we need to compute the actual endpoint type of an endpoint by taking into account the messages in its queue. To this aim we introduce function tail for endpoint types such that

$$\mathsf{tail}(T, \mathtt{m}_1(S_1)\cdots\mathtt{m}_n(S_n)) = T'$$

indicates that messages having tag $\mathtt{m}_i$ and an argument of type S_i can be received in the specified order from an endpoint with type T, which can be used according to type T' thereafter. The function is inductively defined by the following rules:

$$\mathsf{tail}(T, \varepsilon) = T$$

$$\frac{k \in I \qquad S \leqslant S_k}{\mathsf{tail}(\{?\mathtt{m}_i(S_i).T_i\}_{i\in I}, \mathtt{m}_k(S)) = T_k} \qquad \frac{\mathsf{tail}(T, \mathtt{m}(S)) = T'}{\mathsf{tail}(\{\mathsf{e}_i : S_i\}_{i\in I}T, \mathtt{m}(S)) = T'}$$

$$\frac{\mathsf{tail}(T, \mathtt{m}_1(S_1)) = T' \qquad \mathsf{tail}(T', \mathtt{m}_2(S_2)\cdots\mathtt{m}_n(S_n)) = T''}{\mathsf{tail}(T, \mathtt{m}_1(S_1)\mathtt{m}_2(S_2)\cdots\mathtt{m}_n(S_n)) = T''}$$

Notice that $\mathsf{tail}(T, \mathtt{m}(S))$ is undefined when $T = \mathtt{end}$ or T is an internal choice or T denotes the initiation or the termination of a transaction. This will enforce the property that the queue of endpoints having these types must be empty. In the particular case of transaction initiation, this makes sure that, if an exception is thrown, heap restoration simply amounts to emptying the queues of the endpoints involved in the transaction (mechanism (1) in Section 5.1.1). The fact that the queues of the endpoints involved in the transaction are guaranteed to be empty at the end of a transaction is solely motivated by our notion of duality (Definition 5.3.3), which demands a perfect correspondence between the actions on such endpoints during a transaction. In principle, it would be possible to relax duality in such a way that a message sent within a transaction is received only after the transaction is terminated. However, it would still be necessary for the receive operation to first wait for the actual termination of the transaction, for

otherwise the soundness of the transaction would be compromised. This means that this increased flexibility in the syntax of programs would bear no concrete advantage in their semantics.

We now have all the notions to express the well-typedness of a heap μ with respect to a pair $\Gamma_0; \Gamma$ of type environments.

Definition 5.3.10 (well-typed heap). *Let* $\mathsf{dom}(\Gamma_0) \cap \mathsf{dom}(\Gamma) = \emptyset$. *We write* $\Gamma_0; \Gamma \Vdash \mu$ *if all of the following conditions hold:*

(1) *If* $a \mapsto [b, \mathfrak{Q}] \in \mu$ *and* $b \mapsto [a, \mathfrak{Q}'] \in \mu$, *then either* $\mathfrak{Q} = \varepsilon$ *or* $\mathfrak{Q}' = \varepsilon$.

(2) *If* $a \mapsto [b, \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_n(c_n)] \in \mu$, *then*

$$\mathsf{tail}(T, \mathtt{m}_1(S_1) \cdots \mathtt{m}_n(S_n)) = S$$

where $\Gamma_0, \Gamma \vdash a : [\cdots[T]\cdots]$ *and* $\Gamma_0 \vdash c_i : S_i$ *and* $\|S_i\| < \infty$ *and* $\vdash S_i : 0$ *for* $1 \leq i \leq n$ *and* $b \mapsto [a, \varepsilon] \in \mu$ *implies* $\Gamma_0, \Gamma \vdash b : [\cdots[\overline{S}]\cdots]$ *and* $b \notin \mathsf{dom}(\mu)$ *implies* $S = \mathtt{end}$.

(3) $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$;

(4) $A \cap B = \emptyset$ *implies* $\mu\text{-}\mathsf{reach}(A) \cap \mu\text{-}\mathsf{reach}(B) = \emptyset$ *for every* $A, B \subseteq \mathsf{dom}(\Gamma)$.

Condition (1) requires that at least one of the queues of peer endpoints in a well-typed heap is empty. This invariant corresponds to half-duplex communication and is ensured by duality of endpoint types associated with peer endpoints, since a well-typed process cannot send messages on an endpoint until it has read all the pending messages from the corresponding queue (we will see in Example 5.4.12 how to safely circumvent half-duplex communication thanks to transactions). Condition (2) requires that the content of the queue of an endpoint must be consistent with the type of the endpoint, in the sense that the messages in the queue have the expected tag and an argument with the expected type. In addition, the endpoint types of message arguments must all have finite weight and null rank. Finally, the endpoint types of peer endpoints are dual of each other, modulo the content of the non-empty queue. Condition (3) states that the type environment Γ_0, Γ must specify a type for all of the allocated objects in the heap and, in addition, every object (located at) a in the heap must be reachable from a root $b \in \mathsf{dom}(\Gamma)$. Finally, condition (4) requires the uniqueness of the root for every allocated object. Overall, since the roots are distributed linearly among the processes of the system, conditions (3) and (4) guarantee that every allocated object belongs to one and only one process.

There are a few subtleties regarding conditions (1) and (2) and the fact that, in condition (2), the property $b \mapsto [a, \varepsilon] \in \mu$ is the head of an implication. First of all, condition (2) must hold for both peers of a channel, therefore if a is the peer with the empty queue ($n = 0$) while b has messages in its queue, then the type of a is not necessarily the dual of the type of b. The correct dual correspondence is checked when the symmetric pair of endpoints is considered. Second, it is possible that at some point only one endpoint of a channel is allocated. For example, the well-typed process $\mathtt{open}(a, b).\mathtt{close}(b).\mathtt{close}(a)$ reduces to $\mathtt{close}(a)$ in a configuration where the heap contains only $a \mapsto [b, \varepsilon]$. When this happens, the type of the remaining

$$\text{[T-Running Process]}\quad \frac{\Gamma_0; \Gamma_R, \Gamma \Vdash \mu \qquad \tilde{\mathcal{E}}; \Gamma \vdash P}{\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P}$$

$$\text{[T-Running Parallel]}\quad \frac{\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, \Gamma_2; \Gamma_1 \vdash \mu \wr P \qquad \tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, \Gamma_1; \Gamma_2 \vdash \mu \wr Q}{\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma_1, \Gamma_2 \vdash \mu \wr P \mid Q}$$

$$\text{[T-Running Transaction]}\quad \frac{\begin{array}{c}\mu\text{-balanced}(\{a_i : S_{ij}\}_{i\in I})^{\,(j\in J)} \qquad \mu\text{-balanced}(B) \qquad \mathsf{local}(\Gamma_2) \\ \{a_i\}_{i\in I} \cup B = \mu\text{-reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)) \\ \tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \wr P \qquad \tilde{\mathcal{E}}; \Gamma_1, \{a_i : S_{ij}\}_{i\in I} \vdash Q_j\ ^{(j\in J)}\end{array}}{\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2 \vdash \mu \wr \langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle}$$

Figure 5.14: Typing rules for configurations.

endpoint forbids any send operation (last property of condition (2)). Note that condition (1) is not implied by condition (2) and both conditions are necessary.

5.3.6 Typing configurations

Figure 5.14 defines typing rules for configurations $\mu \wr P$ as an extension of the typing rules for processes. Judgments have the form

$$\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P$$

and state that the configuration $\mu \wr P$ is well typed with respect to the exception environment $\tilde{\mathcal{E}}$ and the triple $\Gamma_0; \Gamma_R; \Gamma$ of type environments. Intuitively, Γ is the type environment used to type check P, Γ_R is the type environment describing the type of root pointers owned by processes that are running in parallel with P, and Γ_0 describes the type of pointers that occur in some queue.

Rule [T-Running Process] lifts well-typed processes to well-typed configurations by requiring the heap to be well typed with respect to the pair of environments $\Gamma_0; \Gamma_R, \Gamma$ where Γ_R, Γ represents the whole set of roots obtained from those owned by the process being typed (in Γ) and those owned by processes in parallel with it (in Γ_R).

Rule [T-Running Parallel] is similar to [T-Parallel], except that it deals with three type environments which are appropriately rearranged for keeping track of the roots of the heap.

Rule [T-Running Transaction] captures the basic properties regarding running transactions $\langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : Q_j\}_{j\in J} P\rangle$, which we describe here. The rule makes use of a balancing predicate over type environments that generalizes the notion of balancing for sets of pointers (Definition 5.2.3):

Definition 5.3.11. *We say that* Γ *is* balanced *in* μ*, written* μ-balanced(Γ)*, if* $a \in \mathsf{dom}(\Gamma)$ *and* $a \overset{\mu}{\leftrightarrow} b$ *imply* $b \in \mathsf{dom}(\Gamma)$ *and* $\Gamma(a) = \overline{\Gamma(b)}$.

First of all, it must be possible to partition the type environment into three parts Γ_1, $\{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}$, and Γ_2 such that the environment Γ_1 corresponds to the endpoints owned by P but which are not involved in the transaction. Consequently, the types of these endpoints are sealed in the judgment corresponding to the typing of P. The environment $\{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}$ corresponds to the endpoints involved in the transaction (the first component of the running transaction process), and their type indicates that the transaction is in progress. The environment Γ_2 corresponds to the endpoints that have been allocated inside the transaction. Their type is not sealed in the judgment corresponding to the typing of P. The premises $\mu\text{-}\mathsf{balanced}(\{a_i : S_{ij}\}_{i\in I})$ for every $j \in J$ and $\mu\text{-}\mathsf{balanced}(B)$ indicate that the set of all the endpoints to which P has full access is balanced. Therefore, the transaction operates in a closed scope and cannot have "side effects" from the point of view of other processes. The first premise indicates, in addition, that the types S_{ij} associated with peer endpoints are dual of each other (this property is a consequence of well-typedness of the heap before the transaction initiates, but it must be explicitly stated in [T-Running Transaction] where the heap is checked against a type environment where the S_i's do not occur any more). The premise $\mathsf{local}(\Gamma_2)$ identifies the Γ_2 partition of the context corresponding to the endpoints that have been created inside the transaction. The premise $\{a_i\}_{i\in I} \cup B = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$ states that all the endpoints allocated within the transaction have not escaped the scope of the transaction. The last two premises correspond to the premises of rule [T-Try]. In particular, note that the exception environment is properly augmented when typing the body of the transaction.

Since running transaction processes appear only at runtime as the result of [R-Start Transaction] reductions, they can never occur behind a prefix and therefore the three rules in Figure 5.14 cover all possible forms of runtime configurations.

5.4 Type soundness

We can now formulate the two main results about our framework: well-typedness is preserved by reduction, and well-typed processes are well behaved. The proofs of these theorems require to specify a number of additional properties. The following lemmas say that typing is preserved by structural congruence and by substitutions. In the case of substitutions, subtyping may be applied without compromising well-typedness.

Lemma 5.4.1. *Let* $\tilde{\mathcal{E}}; \Gamma \vdash P$ *and* $P \equiv Q$. *Then* $\tilde{\mathcal{E}}; \Gamma \vdash Q$.

Proof. By case analysis on the derivation of $P \equiv Q$. □

Lemma 5.4.2. *If* $\tilde{\mathcal{E}}; \Gamma \vdash P$ *and* $u \notin \mathsf{dom}(\Gamma)$ *and* $v \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$, *then* $\tilde{\mathcal{E}}; \Gamma \vdash P\{v/x\}$.

Proof. For notational simplicity we prove the result when u is a variable and v is a pointer. Namely, we prove that if $\tilde{\mathcal{E}}; \Gamma \vdash P$ and $x \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$ and $a \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$, then $\tilde{\mathcal{E}}; \Gamma \vdash P\{a/x\}$. If x does not occur free in P then

$P\{a/x\} = P$, so $\tilde{\mathcal{E}}; \Gamma \vdash P\{a/x\}$ trivially holds. We assume $x \in \mathsf{fn}(P)$ and proceed by induction on the derivation of $\tilde{\mathcal{E}}; \Gamma \vdash P$ and by cases on the last rule applied.

[T-Inaction] In this case $\Gamma = \emptyset$ and $P = \mathtt{done}$. So, $P\{a/x\} = \mathtt{done} = P$ and $\tilde{\mathcal{E}}; \Gamma \vdash P\{a/x\}$ holds.

[T-Invoke] In this case $\Gamma = \tilde{u} : \tilde{t}$ and $P = X\langle \tilde{u} \rangle$. From $\mathsf{fn}(X\langle \tilde{u} \rangle) = \tilde{u} = \mathsf{dom}(\Gamma)$, we conclude $x \notin \mathsf{fn}(P)$.

[T-Open] In this case $P = \mathtt{open}(c, d).Q$, $\vdash T : 0$ and $\tilde{\mathcal{E}}; \Gamma, c : T, d : \overline{T} \vdash Q$. We know that $x \notin \{c, d\}$, so by induction hypothesis we obtain $\tilde{\mathcal{E}}; \Gamma, c : T, d : \overline{T} \vdash Q\{a/x\}$. We conclude the proof by an application of [T-Open], since $P\{a/x\} = \mathtt{open}(c, d).Q\{a/x\}$.

[T-Close] In this case $\Gamma = \Gamma', u : \mathtt{end}$, $P = \mathtt{close}(u).Q$, $\tilde{\mathcal{E}}; \Gamma \vdash Q$ and $P\{a/x\} = \mathtt{close}(u).Q\{a/x\}$. We derive the proof by induction hypothesis and [T-Close].

[T-Send] In this case $\Gamma = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, v : S$, $P = u!\mathtt{m}_k(v).Q$, $k \in I$, $\|S\| < \infty$, $S \leqslant S_k$ and $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q$ and $P\{a/x\} = u!\mathtt{m}_k(v).Q\{a/x\}$. By induction hypothesis we obtain $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q\{a/x\}$ and derive the proof by [T-Send].

[T-Receive] In this case $\Gamma = \Gamma', u : \{?\mathtt{m}_i(S_i).T_i\}_{i \in I}$, $P = \sum_{i \in I \cup J} u?\mathtt{m}_i(x_i).P_i$ and $S_i \leqslant S_i'$, and $\tilde{\mathcal{E}}; \Gamma, u : T_i, x_i : S_i' \vdash P_i$ for every $i \in I$. From $\mathsf{dom}(\Gamma') \cup \{u\} \cup \{x_i\} \subseteq \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$, for every $i \in I$ and induction hypothesis we obtain $\tilde{\mathcal{E}}; \Gamma, u : T_i, x_i : S_i' \vdash P_i\{a/x\}$ for every $i \in I$. Since $P\{a/x\} = \sum_{i \in I \cup J} u?\mathtt{m}_i(x_i).P_i\{a/x\}$, we derive the proof from [T-Receive]. This case is, actually, the only interesting one. The reason is that, because of subtyping, we have a premise for every $i \in I$ and the process is the summation for $i \in I \cup J$.

[T-Choice] In this case $P = P_1 \oplus P_2$, $\tilde{\mathcal{E}}; \Gamma \vdash P_1$ and $\tilde{\mathcal{E}}; \Gamma \vdash P_2$. We derive the proof by induction hypothesis and [T-Choice].

[T-Parallel] In this case $\Gamma = \Gamma_1, \Gamma_2$, $P = P_1 \,|\, P_2$, $\tilde{\mathcal{E}}; \Gamma_1 \vdash P_1$ and $\tilde{\mathcal{E}}; \Gamma_2 \vdash P_2$. We derive the proof by induction hypothesis and [T-Parallel].

[T-Try] In this case $\Gamma = \Gamma', \{u_i : \{\mathsf{e}_j : S_{ij}\}_{j \in J}[\![T_i]\!]_{i \in I}$, $P = \mathtt{try}(\{u_i\}_{i \in I})\ \{\mathsf{e}_j : R_j\}_{j \in J} Q$, $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; [\Gamma'], \{u_i : T_i\}_{i \in I} \vdash Q$ and $\tilde{\mathcal{E}}; \Gamma', \{u_i : S_{ij}\}_{i \in I} \vdash R_j$ for every $j \in J$. Since $P\{a/x\} = \mathtt{try}(\{u_i\}_{i \in I})\ \{\mathsf{e}_j : R_j\{a/x\}\}_{j \in J} Q\{a/x\}$, we derive the proof by induction hypothesis and [T-Try].

[T-Throw] In this case $P = \mathtt{throw}\ \mathsf{e}$. So, $P\{a/x\} = P$.

[T-Commit] In this case $\tilde{\mathcal{E}} = \tilde{\mathcal{E}}'\mathcal{E}$, $\Gamma = [\Gamma_1], \{u_i : [\![T_i]\!]_{i \in I}, \Gamma_2$, $P = \mathtt{commit}(\{u_i\}_{i \in I}).Q$ and $\tilde{\mathcal{E}}'; \Gamma_1, \{u_i : T_i\}_{i \in I}, \Gamma_2 \vdash Q$. Since $P\{a/x\} = \mathtt{commit}(\{u_i\}_{i \in I}).Q\{a/x\}$, we derive the proof by induction hypothesis and [T-Commit]. □

Lemma 5.4.3 (Substitution). *If* $\tilde{\mathcal{E}}; \Gamma, u : t \vdash P$ *and* $v \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$ *and* $s \leqslant t$, *then* $\tilde{\mathcal{E}}; \Gamma, v : s \vdash P\{v/u\}$.

Proof. For notational simplicity we prove the result when u is a variable and v is a pointer. Namely, we prove that if $\tilde{\mathcal{E}}; \Gamma, x : t \vdash P$ and $a \notin \mathsf{dom}(\Gamma) \cup \mathsf{bn}(P)$,

and $s \leqslant t$, then $\tilde{\mathcal{E}}; \Gamma, a : s \vdash P\{a/x\}$. We proceed by induction on the derivation of $\tilde{\mathcal{E}}; \Gamma, x : t \vdash P$ and by cases on the last rule applied. We only prove a few interesting cases.

[T-Inaction] This case is impossible.

[T-Open] In this case $P = \texttt{open}(c, d).Q$. Since we know $x \notin \{c, d\}$ the proof is concluded by induction.

[T-Close] In this case $P = \texttt{close}(u).Q$ and $\Gamma, x : t = \Gamma', u : \texttt{end}$ and $\tilde{\mathcal{E}}; \Gamma' \vdash Q$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'', x : t$ and $\Gamma = \Gamma'', u : \texttt{end}$. By induction hypothesis we deduce $\tilde{\mathcal{E}}; \Gamma'', a : s \vdash Q\{a/x\}$. Since we know that $a \neq u$, from rule [T-Close] we conclude $\tilde{\mathcal{E}}; \Gamma, a : s \vdash \texttt{close}(u).Q\{a/x\}$.

If $x = u$, then $\Gamma = \Gamma'$, $t = \texttt{end}$. By Lemma 5.4.2, we have $\tilde{\mathcal{E}}; \Gamma \vdash Q\{a/x\}$. From rule [T-Close] we conclude $\tilde{\mathcal{E}}; \Gamma, a : \texttt{end} \vdash \texttt{close}(a).Q\{a/x\}$.

[T-Send] In this case $\Gamma, x : t = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, v : S$ and $P = u!\mathtt{m}_k(v).Q$ where $k \in I$ and $\|S\| < \infty$ and $S \leqslant S_k$ and $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'', x : t$ and $\Gamma = \Gamma'', u : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, v : S_k$. From $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q$ and by induction hypothesis we obtain $\tilde{\mathcal{E}}; \Gamma'', a : s, u : T_k \vdash Q\{a/x\}$. Since $a \notin \mathsf{dom}(\Gamma)$ we know that $a \notin \{u, v\}$ and from rule [T-Send] we conclude $\tilde{\mathcal{E}}; \Gamma, a : s \vdash u!\mathtt{m}_k(v).Q\{a/x\}$.

If $x = u$, then $\Gamma = \Gamma', v : S.$ and $t = \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}$ and $s = \{!\mathtt{m}_i(S'_i).T'_i\}_{i \in I \cup J}$ and $S_i \leqslant S'_i$ and $T'_i \leqslant T_i$ for every $i \in I$. Then $S \leqslant S_k \leqslant S'_k$. From $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q$ and the induction hypothesis we obtain $\tilde{\mathcal{E}}; \Gamma', a : T_k \vdash Q\{a/x\}$. We conclude $\tilde{\mathcal{E}}; \Gamma', a : \{!\mathtt{m}_i(S'_i).T'_i\}_{i \in I}, v : S \vdash a!\mathtt{m}_k(v).Q\{a/x\}$ with an application of rule [T-Send].

If $x = v$, then $t = S$ and $x \notin \mathsf{dom}(\Gamma') \cup \{u\}$, so by Lemma 5.4.2 $\tilde{\mathcal{E}}; \Gamma', u : T_k \vdash Q\{a/x\}$. From Proposition 5.3.8 we deduce $\|s\| \leq \|t\| < \infty$. We conclude with an application of rule [T-Send].

[T-Try] In this case $\Gamma, x : t = \Gamma', \{u_i : \{\mathsf{e}_j : S_{ij}\}_{j \in J}[\![T_i\}_{i \in I}$ and $P = \texttt{try}(\{u_i\}_{i \in I})\ \{\mathsf{e}_j : Q_j\}_{j \in J} Q$ and $\tilde{\mathcal{E}}; \{\mathsf{e}_j\}_{j \in J}; [\Gamma'], \{u_i : T_i\}_{i \in I} \vdash Q$ and $\tilde{\mathcal{E}}; \Gamma', \{u_i : S_{ij}\}_{i \in I} \vdash Q_j$ for every $j \in J$.

If $x \in \mathsf{dom}(\Gamma')$, then $\Gamma' = \Gamma'', x : t$ for some Γ''. From the hypothesis $s \leqslant t$ we deduce $[s] \leqslant [t]$. From $\tilde{\mathcal{E}}; \{\mathsf{e}_j\}_{j \in J}; [\Gamma'], \{u_i : T_i\}_{i \in I} \vdash Q$ and by induction hypothesis we deduce $\tilde{\mathcal{E}}; \{\mathsf{e}_j\}_{j \in J}; [\Gamma''], a : [s], \{u_i : T_i\}_{i \in I} \vdash Q\{a/x\}$. From $\tilde{\mathcal{E}}; \Gamma', \{u_i : S_{ij}\}_{i \in I} \vdash Q_j$ and by induction hypothesis we deduce $\tilde{\mathcal{E}}; \Gamma'', a : s, \{u_i : S_{ij}\}_{i \in I} \vdash Q_j\{a/x\}$ for every $j \in J$. We conclude with an application of rule [T-Try].

If $x = u_k$ for some $k \in I$, then $t = \{\mathsf{e}_j : S_{kj}\}_{j \in J}[\![T_k$ and $s = \{\mathsf{e}_j : S'_{kj}\}_{j \in J}[\![T'_k$ where $S'_{kj} \leqslant S_{kj}$ for every $j \in J$ and $T'_k \leqslant T_k$. By induction hypothesis we deduce $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; [\Gamma'], \{u_i : T_i\}_{i \in I \setminus \{k\}}, a : T'_k \vdash Q\{a/x\}$ and $\tilde{\mathcal{E}}; \Gamma', \{u_i : S_{ij}\}_{i \in I \setminus \{k\}}, a : S'_{kj} \vdash Q_j\{a/x\}$ for every $j \in J$. We conclude with an application of rule [T-Try]. □

The next lemma connects well-typed configurations to well-typed heaps and shows the irrelevance of the first and second components Γ_0 and Γ_R in typing processes.

Lemma 5.4.4. *Let* $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \triangleright P$. *Then:*

(1) $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$;

(2) $\Gamma'_0; \Gamma'_R, \Gamma \Vdash \mu'$ *implies* $\tilde{\mathcal{E}}; \Gamma'_0; \Gamma'_R; \Gamma \vdash \mu' \triangleright P$.

Proof. By induction on the derivation of $\Gamma_0; \Gamma_R; \Gamma \vdash \mu \triangleright P$ and by cases on the last rule applied. The only interesting case is [T-Running Transaction], from which we deduce:

- $P = \langle \{a_i\}_{i \in I}, B, \{\mathsf{e}_j : R_j\}_{j \in J} Q \rangle$;
- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j \in J} T_i\}_{i \in I}, \Gamma_2$;
- μ-$\mathsf{balanced}(\{a_i : S_{ij}\}_{i \in I})$ for every $j \in J$;
- $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \vdash \mu \triangleright Q$.

Regarding (1), from $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \vdash \mu \triangleright Q$ by induction hypothesis we obtain $\Gamma_0; \Gamma_R, [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \Vdash \mu$ and then from the definition of function tail and μ-$\mathsf{balanced}(\{a_i : S_{ij}\}_{i \in I})$ for $j \in J$ we conclude $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$.

Regarding (2), from $\Gamma'_0; \Gamma'_R, \Gamma \Vdash \mu'$ and the definition of function tail we deduce $\Gamma'_0; \Gamma'_R, [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \Vdash \mu'$ and then from $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \vdash \mu \triangleright Q$ by induction hypothesis we obtain $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma'_0; \Gamma'_R; [\Gamma_1], \{a_i : T_i\}_{i \in I}, \Gamma_2 \vdash \mu' \triangleright Q$. We conclude with an application of [T-Running Transaction]. □

The next lemma shows that the rule [T-Running Parallel] for configurations subsumes the rule [T-Parallel] for processes. It is used for simplifying some cases in the proof of subject reduction (Theorem 5.4.7).

Lemma 5.4.5. *If* $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \triangleright P_1 \mid P_2$ *is derivable using* [T-Parallel] *and* [T-Running Process]*, then it is also derivable using* [T-Running Process] *and* [T-Running Parallel].

Proof. From $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \triangleright P_1 \mid P_2$ and rule [T-Running Process] we obtain (H.1) $\tilde{\mathcal{E}}; \Gamma \vdash P_1 \mid P_2$ and (H.2) $\Gamma_0; \Gamma_R, \Gamma \Vdash \mu$. From (H.1) and rule [T-Parallel] we obtain (T.1) $\Gamma = \Gamma_1, \Gamma_2$ and (P.i) $\tilde{\mathcal{E}}; \Gamma_i \vdash P_i$ for $i \in \{1, 2\}$. From (H.2), (T.1), (P.i) and rule [T-Running Process] we obtain $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, \Gamma_{3-i}; \Gamma_i \vdash \mu \triangleright P_i$ for $i \in \{1, 2\}$. We conclude with an application of rule [T-Running Parallel]. □

When a new channel is allocated in the heap, it always comes as a pair of peer endpoints. This easy property is formalized thus:

Proposition 5.4.6. *If* $\mu \triangleright P \to \mu' \triangleright P'$ *then* μ'-$\mathsf{balanced}(\mathsf{dom}(\mu') \setminus \mathsf{dom}(\mu))$.

Proof. Simple induction on the reduction that occurs. □

Subject reduction takes into account the possibility that types in the environment may change as the process reduces, which is common in behavioral type theories. The next result is the complete version of subject reduction proving that reductions preserve well-typedness and showing the relationship between the contexts used for typing the two configurations involved. In particular, item (1) below says that reductions preserve well-typedness; item (2) says that the rank

of the type of endpoints are preserved by reductions, and that (de)allocated endpoints have a type with null rank; finally, item (3) formally expresses the concept of process isolation, saying that any portion of the heap that is not reachable by the process being reduced is not affected by the reduction. In this theorem and in its proof, we write $\mathsf{unsealed}(\Gamma)$ if all types in the range of Γ are unsealed. We also write $\bigsqcup_{i\in I}\Gamma_i$ for the disjoint union of the contexts Γ_i.

Theorem 5.4.7 (Subject reduction). *Let* $\tilde{\mathcal{E}};\Gamma_0;\Gamma_R;[\Gamma_S],\Gamma \vdash \mu \fatsemi P$ *where* $\mathsf{unsealed}(\Gamma)$ *and* $\mu \fatsemi P \to \mu' \fatsemi P'$. *Then there exist* Γ_0' *and* Γ' *such that:*

(1) $\tilde{\mathcal{E}};\Gamma_0';\Gamma_R;[\Gamma_S],\Gamma' \vdash \mu' \fatsemi P'$, *and*

(2) $\mathsf{unsealed}(\Gamma')$ *and for every* $a \in \mathsf{dom}(\Gamma)\cap\mathsf{dom}(\Gamma')$ *we have* $\mathsf{rank}(\Gamma(a)) = \mathsf{rank}(\Gamma'(a))$ *and for every* $a \in \mathsf{dom}(\Gamma)\setminus\mathsf{dom}(\Gamma')$ *we have* $\mathsf{rank}(\Gamma(a)) = 0$ *and for every* $a \in \mathsf{dom}(\Gamma')\setminus\mathsf{dom}(\Gamma)$ *we have* $\mathsf{rank}(\Gamma'(a)) = 0$, *and*

(3) for every $\Gamma_I \subseteq \Gamma_R, [\Gamma_S]$ *such that* $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I,\Gamma)))$ *we have*

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R,\Gamma_S)\setminus\mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R,\Gamma_S)\setminus\mathsf{dom}(\Gamma_I)).$$

Proof. By induction on the derivation of $\mu \fatsemi P \to \mu' \fatsemi P'$ and by cases on the last rule applied. We omit trivial and symmetric cases.

[R-Open] Then $P = \mathtt{open}(a,b).P'$ and $\mu' = \mu, a \mapsto [b,\varepsilon], b \mapsto [a,\varepsilon]$. From rule [T-Running Process] we obtain:

- (H.1) $\tilde{\mathcal{E}};[\Gamma_S],\Gamma \vdash \mathtt{open}(a,b).P'$;
- (H.2) $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma \Vdash \mu$.

From (H.1) and rule [T-Open] we obtain:

- $\vdash T : 0$;
- (C.1) $\tilde{\mathcal{E}};[\Gamma_S],\Gamma,a:T,b:\overline{T} \vdash P'$.

Let $\Gamma_0' = \Gamma_0$ and $\Gamma' = \Gamma, a:T, b:\overline{T}$. The proof of (C.2) $\Gamma_0';\Gamma_R,[\Gamma_S],\Gamma' \Vdash \mu'$ is trivial. From (C.1), (C.2) and [T-Running Process] we obtain (1). We conclude by noting that items (2) and (3) hold trivially.

[R-Close] In this case $P = \mathtt{close}(a).P'$ and $\mu = \mu', a \mapsto [b,\mathfrak{Q}]$. From rule [T-Running Process] we obtain:

- (H.1) $\tilde{\mathcal{E}};[\Gamma_S],\Gamma \vdash \mathtt{close}(a).P'$;
- (H.2) $\Gamma_0;\Gamma_R,[\Gamma_S],\Gamma \Vdash \mu', a \mapsto [b,\mathfrak{Q}]$.

From the hypothesis (H.1) and rule [T-Close] we obtain:

- (L.1) $\Gamma = \Gamma', a : \mathtt{end}$;
- (C.1) $\tilde{\mathcal{E}};[\Gamma_S],\Gamma' \vdash P'$.

Let $\Gamma'_0 = \Gamma_0$. We only have to show that (C.2) $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and the only interesting case in Definition 5.3.10 is item 3. The relation $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma'_0, \Gamma_R, [\Gamma_S], \Gamma')$ is obvious. We need to prove

$$\mathsf{dom}(\Gamma'_0, \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma'_R, [\Gamma_S], \Gamma')).$$

First we show that $\mathfrak{Q}$ is empty. Suppose by contradiction that this is not the case. Then the endpoint type associated with a before the reduction occurs must begin with an external choice or running transaction, which contradicts (L.1). So we obtain

$$\begin{array}{rcll} \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma')) & = & \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma)) \setminus \{a\} & \\ & = & \mathsf{dom}(\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma) \setminus \{a\} & \text{(H.2)} \\ & = & \mathsf{dom}(\Gamma'_0, \Gamma_R, \Gamma') & \end{array}$$

From (C.1), (C.2) and [T-Running Process] we conclude (1). Item (2) is obvious while item (3) holds because $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S))$.

[R-Parallel] In this case $P = P_1 \mid P_2$ and $\mu \triangleright P_1 \to \mu' \triangleright P'_1$ and $P' = P'_1 \mid P_2$. By Lemma 5.4.5 we can assume that $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \vdash \mu \triangleright P$ was derived by an application of rule [T-Running Parallel]. Then:

- $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_S = \Gamma_{S1}, \Gamma_{S2}$;
- (P.i) $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, [\Gamma_{S3-i}], \Gamma_{3-i}; [\Gamma_{Si}], \Gamma_i \vdash \mu \triangleright P_i$ for $i \in \{1, 2\}$.

From (P.1) by induction hypothesis we deduce that there exist Γ'_0 and Γ'_1 such that:

(1') $\tilde{\mathcal{E}}; \Gamma'_0; \Gamma_R, [\Gamma_{S2}], \Gamma_2; [\Gamma_{S1}], \Gamma'_1 \vdash \mu' \triangleright P'_1$, and

(2') $\mathsf{unsealed}(\Gamma'_1)$ and for every $a \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma'_1)$ we have $\mathsf{rank}(\Gamma_1(a)) = \mathsf{rank}(\Gamma'_1(a))$ and for every $a \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma'_1)$ we have $\mathsf{rank}(\Gamma_1(a)) = 0$ and for every $a \in \mathsf{dom}(\Gamma'_1) \setminus \mathsf{dom}(\Gamma_1)$ we have $\mathsf{rank}(\Gamma'_1(a)) = 0$, and

(3') for every $\Gamma_I \subseteq \Gamma_R, [\Gamma_S], \Gamma_2$ such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_1)))$ we have

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)).$$

Let $\Gamma' = \Gamma'_1, \Gamma_2$. From (1') and Lemma 5.4.4(1) we obtain (N.1) $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. From (P.2), (N.1), and Lemma 5.4.4(2) we deduce (P.2') $\tilde{\mathcal{E}}; \Gamma'_0; \Gamma_R, [\Gamma_{S1}], \Gamma'_1; [\Gamma_{S2}], \Gamma_2 \vdash \mu' \triangleright P_2$. From (1'), (P.2'), and rule [T-Running Parallel] we conclude (1). Regarding (2), just notice that $\mathsf{unsealed}(\Gamma')$. Regarding (3), let $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ be such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_J, \Gamma)))$. Take $\Gamma_I = \Gamma_J, \Gamma_2$ and observe that $\Gamma_I \subseteq \Gamma_R, [\Gamma_S], \Gamma_2$ and $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_1)))$. From (3') we are able to deduce $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I))$ and we conclude (3) by observing that $\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_J) = \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_2) \setminus \mathsf{dom}(\Gamma_I)$.

[R-Send] In this case $P = a!\mathtt{m}(c).P'$ and $\mu = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$ and $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}' :: \mathtt{m}(c)]$. From [T-Running Process] we obtain:

- (H.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma \vdash a!\mathtt{m}(c).P'$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu'', a \mapsto [b, \mathfrak{Q}], b \mapsto [a, \mathfrak{Q}']$.

From (H.1) and rule [T-Send] we deduce:

- (L.1) $\Gamma = \Gamma'', a : \{!\mathtt{m}_i(S_i).T_i\}_{i \in I}, c : S$;
- $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$;
- $S \leqslant S_k$ and $\|S\| < \infty$;
- (C.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma'', a : T_k \vdash P'$.

Let $\Gamma_0' = \Gamma_0, c : S_k$ and $\Gamma' = \Gamma'', a : T_k$. We show (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ by proving the items of Definition 5.3.10 in order.

1. We only need to show that $\mathfrak{Q}$ is empty. Suppose by contradiction that this is not the case. Then the endpoint type associated with a before the reduction must begin with an external choice or running transaction, which contradicts (L.1).

2. Let $\mathfrak{Q}' = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$. From hypothesis (H.2) we deduce $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash b : T_b$ and $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash c_i : S_i'$ for $1 \leq i \leq p$ where

$$\mathsf{tail}(T_b, \mathtt{m}_1(S_1') \cdots \mathtt{m}_p(S_p')) = \overline{\{!\mathtt{m}_i(S_i).T_i\}_{i \in I}} = \{?\mathtt{m}_i(S_i).\overline{T_i}\}_{i \in I}$$

   and from $S \leqslant S_k$ we conclude $\overline{T_k} = \mathsf{tail}(T_b, \mathtt{m}_1(S_1') \cdots \mathtt{m}_p(S_p')\mathtt{m}(S))$.

3. From hypothesis (H.2) we have $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma_R, \Gamma)$ and for every $a' \in \mathsf{dom}(\mu)$ there exists $b' \in \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma)$ such that $a' \preccurlyeq_\mu b'$. Clearly $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma_0', \Gamma_R, \Gamma_S, \Gamma')$ since $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$ and $\mathsf{dom}(\Gamma_0') \cup \mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma_0) \cup \mathsf{dom}(\Gamma)$. Let $b \preccurlyeq_\mu b_0$ and $\Gamma_R, \Gamma_S, \Gamma \vdash b_0 : T_0$. We have $c \prec_{\mu'} b \preccurlyeq_{\mu'} b_0$, namely $c \preccurlyeq_{\mu'} b_0$. Now

$$\|S\| \leq \|S_k\| < \|\mathsf{tail}(T_b, \mathtt{m}_1(S_1') \cdots \mathtt{m}_p(S_p'))\| \leq \|T_b\| \leq \|T_0\|$$

   and $\|S\| \leq \infty$, therefore $c \neq b_0$. We conclude $b_0 \in \mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma')$.

4. Immediate from hypothesis (H.2).

Item (2) holds trivially. Regarding item (3), let $\Gamma_I \subseteq \Gamma_R, [\Gamma_S]$ be such that $\mu\text{-}\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma)))$. From $a \in \mathsf{dom}(\Gamma)$ we are able to deduce that $b \in \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma))$ and $c \preccurlyeq_{\mu'} b$, therefore $\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_I))$.

[R-Receive] In this case $P = \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$ and $\mu = \mu'', a \mapsto [b, \mathtt{m}(c) :: \mathfrak{Q}]$ where $\mathfrak{Q} = \mathtt{m}_1(c_1) :: \cdots :: \mathtt{m}_p(c_p)$ and $\mathtt{m} = \mathtt{m}_k$ for some $k \in I$ and $P' = P_k\{c/x_k\}$ and $\mu' = \mu'', a \mapsto [b, \mathfrak{Q}]$. From rule [T-Running Process] we obtain:

- (H.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma \vdash \sum_{i \in I} a?\mathtt{m}_i(x_i).P_i$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$.

From (H.1) and rule [T-Receive] we obtain:

- $\Gamma = \Gamma'', a : \{?\mathtt{m}_i(S_i).T_i\}_{i\in J}$ with $J \subseteq I$;
- (N.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma'', a : T_k, x_k : S_k \vdash P_k$.

From (H.2) we deduce $\Gamma_0 = \Gamma_0', c : S$ where $S \leqslant S_k$ and $k \in J$. Let $\Gamma' = \Gamma'', a : T_k, c : S$. From (N.1) and Lemma 5.4.3 we deduce (C.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma' \vdash P_k\{c/x_k\}$. Now we only have to show (C.2) $\Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and we do it by proving the items of Definition 5.3.10 in order.

1. Since the queue associated with a is not empty in μ, the queue associated with its peer endpoint b must be empty. The reduction does not change the queue associated with b, therefore condition (1) of Definition 5.3.10 is satisfied.
2. From (H.2) we deduce $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash b : T_b$ and
$$\overline{T_b} = \mathsf{tail}(\{?\mathtt{m}_i(S_i).T_i\}_{i\in J}, \mathtt{m}(S)\mathtt{m}_1(S_1') \cdots \mathtt{m}_p(S_p')) = \mathsf{tail}(T_k, \mathtt{m}_1(S_1') \cdots \mathtt{m}_p(S_p'))$$
where $\Gamma_0, \Gamma_R, [\Gamma_S], \Gamma \vdash c_i : S_i'$ for $1 \leq i \leq p$.
3. Straightforward by definition of Γ_0' and Γ'.
4. Immediate from (H.2).

Therefore, from (C.1), (C.2), and rule [T-Running Process] we conclude $\tilde{\mathcal{E}}; \Gamma_0'; \Gamma_R, [\Gamma_S], \Gamma' \vdash \mu' \wr P'$. Regarding (2), observe from (H.2) and condition (2) of Definition 5.3.10 that $\mathsf{rank}(S_k) = 0$. Regarding (3), it suffices to observe that the only region of the heap that changes is the queue associated with a and that $a \notin \mathsf{dom}(\Gamma_R, \Gamma_S)$.

[R-Start Transaction] In this case $P = \prod_{i\in I} \mathtt{try}(A_i)\ \{\mathsf{e}_j : Q_{ij}\}_{j\in J} P_i$ where $\mu\text{-}\mathsf{balanced}(\bigcup_{i\in I} A_i)$ and $P' = \langle \bigcup_{i\in I} A_i, \emptyset, \{\mathsf{e}_j : \prod_{i\in I} Q_{ij}\}_{j\in J} \prod_{i\in I} P_i\rangle$ and $\mu' = \mu$. According to Lemma 5.4.5 we can assume that $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; [\Gamma_S], \Gamma \vdash \mu \wr P$ was derived by rule [T-Running Parallel]. Then:

- (L.1) $\Gamma_S = \bigsqcup_{i\in I} \Gamma_{Si}$ and $\Gamma = \bigsqcup_{i\in I} \Gamma_i$;
- (P.i) $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R, \bigsqcup_{j\in I\setminus\{i\}} [\Gamma_{Sj}], \bigsqcup_{j\in I\setminus\{i\}} \Gamma_j; [\Gamma_{Si}], \Gamma_i \vdash \mu \wr \mathtt{try}(A_i)\ \{\mathsf{e}_j : Q_{ij}\}_{j\in J} P_i$ for every $i \in I$.

From (L.1), (P.i) and rule [T-Running Process] we obtain:

- (H.1) $\tilde{\mathcal{E}}; [\Gamma_{Si}], \Gamma_i \vdash \mathtt{try}(A_i)\ \{\mathsf{e}_j : Q_{ij}\}_{j\in J} P_i$ where $\mathsf{unsealed}(\Gamma_i)$ for every $i \in I$;
- (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$.

From (H.1) and rule [T-Try] we obtain, for every $i \in I$:

- (L.2) $\Gamma_i = \Gamma_i', \{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J} [\![T_a\}_{a\in A_i}$;
- $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j\in J}; [[\Gamma_{Si}], \Gamma_i'], \{a : T_a\}_{a\in A_i} \vdash P_i$;

- $\tilde{\mathcal{E}}; [\Gamma_{Si}], \Gamma'_i, \{a : S_{aj}\}_{a \in A_i} \vdash Q_{ij}$ for every $j \in J$.

By rule [T-Parallel] we derive:

- (P.1) $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; [[\Gamma_S]], \bigsqcup_{i \in I} \Gamma'_i], \{a : T_a\}_{i \in I, a \in A_i} \vdash \prod_{i \in I} P_i$;
- (P.2) $\tilde{\mathcal{E}}; [\Gamma_S], \bigsqcup_{i \in I} \Gamma'_i, \{a : S_{aj}\}_{i \in I, a \in A_i} \vdash \prod_{i \in I} Q_{ij}$ for every $j \in J$.

From (L.2) and the definition of tail it is easy to see that the queue associated with a is empty for every $i \in I$ and $a \in A_i$. Hence, from (H.2), (L.1), and (L.2) and the fact that $\mu\text{-}\mathsf{balanced}(\bigcup_{i \in I} A_i)$ we have $\mu\text{-}\mathsf{balanced}(\{a : S_{aj}\}_{i \in I, a \in A_i})$ for every $j \in J$ and $\bigcup_{i \in I} A_i = \mu\text{-}\mathsf{reach}(\bigcup_{i \in I} A_i)$. Also, from (H.2) we have

- (H.2') $\Gamma_0; \Gamma_R, [[\Gamma_S]], \bigsqcup_{i \in I} \Gamma'_i], \{a : T_a\}_{i \in I, a \in A_i} \Vdash \mu$.

From (H.2'), (P.1), and rule [T-Running Process] we obtain:

- (T.1) $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma_0; \Gamma_R; [[\Gamma_S]], \bigsqcup_{i \in I} \Gamma'_i], \{a : T_a\}_{i \in I, a \in A_i} \vdash \mu \fatsemi \prod_{i \in I} P_i$.

We conclude (1) with an application of rule [T-Running Transaction], (T.1), (P.2), and the facts proven above by taking $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \bigsqcup_{i \in I} \Gamma'_i, \{a : \{\mathsf{e}_j : S_{aj}\}_{j \in J} T_a\}_{i \in I, a \in A_i}$. Item (2) is trivial and (3) holds since $\mu' = \mu$.

[R-End Transaction] Then $P = \langle A, B, \{\mathsf{e}_j : Q_j\}_{j \in J} \prod_{i \in I} \mathtt{commit}(A_i).P_i \rangle$ and $P' = \prod_{i \in I} P_i$ and $\mu' = \mu$. From rule [T-Running Transaction] we obtain:

- $\Gamma = \Gamma_1, \{a : \{\mathsf{e}_j : S_{aj}\}_{j \in J} T_a\}_{a \in A}, \Gamma_2$;
- (T.1) $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; \Gamma_0; \Gamma_R; [[\Gamma_S]], \Gamma_1], \{a : T_a\}_{a \in A}, \Gamma_2 \vdash \mu \fatsemi \prod_{i \in I} \mathtt{commit}(A_i).P_i$;
- $\mu\text{-}\mathsf{balanced}(\{a : S_{aj}\}_{a \in A})$ for every $j \in J$;
- $\mu\text{-}\mathsf{balanced}(B)$;
- $\mathsf{local}(\Gamma_2)$;
- $A \cup B = \mu\text{-}\mathsf{reach}(A \cup \mathsf{dom}(\Gamma_2))$.

From (T.1), [T-Running Parallel], and [T-Running Process] we deduce:

- $\Gamma_S = \bigsqcup_{i \in I} \Gamma_{Si}$ and $\Gamma_1 = \bigsqcup_{i \in I} \Gamma_{1i}$ and $\{a : T_a\}_{a \in A} = \bigsqcup_{i \in I} \{a : T_a\}_{a \in B_i}$ and $\Gamma_2 = \bigsqcup_{i \in I} \Gamma_{2i}$ where $\mathsf{rank}(\Gamma_{2i}) = 0$ for every $i \in I$;
- $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j \in J}; [[\Gamma_{Si}], \Gamma_{1i}], \{a : T_a\}_{a \in B_i}, \Gamma_{2i} \vdash \mathtt{commit}(A_i).P_i$ for every $i \in I$.

From rule [T-Commit] we deduce:

- $B_i = A_i$;
- $T_a = \rrbracket T'_a$ for every $i \in I$ and $a \in A_i$;
- $\tilde{\mathcal{E}}; [\Gamma_{Si}], \Gamma_{1i}, \{a : T'_a\}_{a \in A_i}, \Gamma_{2i} \vdash P_i$ for every $i \in I$.

From rule [T-Parallel] we deduce (H.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma_1, \{a : T'_a\}_{i\in I, a\in A_i}, \Gamma_2 \vdash P'$. Let $\Gamma'_0 = \Gamma_0$ and $\Gamma' = \Gamma_1, \{a : T'_a\}_{a\in A}, \Gamma_2$. From $\tilde{\mathcal{E}}; \Gamma_0; \Gamma_R; [\Gamma_S], \Gamma \vdash \mu \wr P$ and Lemma 5.4.4(1) we obtain (H.2) $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma \Vdash \mu$. From (2) we deduce that the queues associated with the pointers $a \in A$ are empty, because Γ includes $\{a : \{\mathsf{e}_j : S_{aj}\}_{j\in J}]\!]T'_a\}_{a\in A}$. Hence we deduce $\Gamma_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$. From (H.1) and rule [T-Running Process] we conclude (1). We observe that (2) holds since $\mathsf{rank}(\{\mathsf{e}_j : S_{aj}\}_{j\in J}T_a) = \mathsf{rank}(T'_a)$ for $a \in A$ and (3) holds trivially since the heap has not changed.

[R-Run Transaction] In this case: $P = \langle A, B, \{\mathsf{e}_j : R_j\}_{j\in J}Q\rangle$ and $P' = \langle A, B', \{\mathsf{e}_j : R_j\}_{j\in J}Q'\rangle$ where $B' = \mathsf{track}(B, \mathsf{dom}(\mu), \mathsf{dom}(\mu'))$ and $\mu \wr Q \to \mu' \wr Q'$. Let $A = \{a_i\}_{i\in I}$. From rule [T-Running Transaction] we deduce:

- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J}T_i\}_{i\in I}, \Gamma_2$;
- (T.1) $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \wr Q$;
- (T.2) $\tilde{\mathcal{E}}; \Gamma_1, \{a_i : S_{ij}\}_{i\in I} \vdash R_j$ for every $j \in J$;
- (T.3) μ-$\mathsf{balanced}(\{a_i : S_{ij}\}_{i\in I})$;
- (T.4) μ-$\mathsf{balanced}(B)$;
- (T.5) $\mathsf{local}(\Gamma_2)$;
- (T.6) $\{a_i\}_{i\in I} \cup B = \mu$-$\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$.

Let $\Gamma_3 = \{a_i : T_i\}_{i\in I}, \Gamma_2$. From (T.1) and $\mathsf{unsealed}(\Gamma)$ by induction hypothesis we obtain that there exist Γ'_0 and Γ'_3 such that:

(1') $\tilde{\mathcal{E}}; \Gamma'_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma'_3 \vdash \mu' \wr Q'$, and

(2') $\mathsf{unsealed}(\Gamma'_3)$ and for every $a \in \mathsf{dom}(\Gamma_3) \cap \mathsf{dom}(\Gamma'_3)$ we have $\mathsf{rank}(\Gamma_3(a)) = \mathsf{rank}(\Gamma'_3(a))$ and for every $a \in \mathsf{dom}(\Gamma_3) \setminus \mathsf{dom}(\Gamma'_3)$ we have $\mathsf{rank}(\Gamma_3(a)) = 0$ and for every $a \in \mathsf{dom}(\Gamma'_3) \setminus \mathsf{dom}(\Gamma_3)$ we have $\mathsf{rank}(\Gamma'_3(a)) = 0$, and

(3') for every $\Gamma_I \subseteq \Gamma_R, [[\Gamma_S], \Gamma_1]$ such that μ-$\mathsf{balanced}(\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_I, \Gamma_3)))$ we have

$$\mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)).$$

Since $\mathsf{rank}(T_i) > 0$ for all $i \in I$, from (2') we deduce that all the a_i's are still in the environment for Q'. Therefore we have $\Gamma'_3 = \{a : T'_i\}_{i\in I}, \Gamma'_2$. Let $\Gamma' = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J}T'_i\}_{i\in I}, \Gamma'_2$.

Regarding (1), from (T.4) and Proposition 5.4.6 we obtain (T.4') μ'-$\mathsf{balanced}(B')$. From (2') we deduce (T.5') $\mathsf{local}(\Gamma'_2)$. In order to prove (T.6') $\{a_i\}_{i\in I} \cup B' =$

μ'-reach$(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma'_2))$ observe that:

$$
\begin{array}{ll}
(*)\ \mathsf{dom}(\mu') & \\
\quad = (\mathsf{dom}(\mu) \cup (B' \setminus B)) \setminus (B \setminus B') & \text{by definition of } B' \\
\quad = (\mu\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma_2) \cup \{a_i\}_{i\in I}) \cup (B' \setminus B)) \setminus (B \setminus B') & \\
\qquad\qquad \text{by item (3) of Definition 5.3.10} & \\
\quad = (\mu\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) & \\
\qquad \uplus\, \mu\text{-reach}(\mathsf{dom}(\Gamma_2) \cup \{a_i\}_{i\in I}) \cup (B' \setminus B)) \setminus (B \setminus B') & \\
\qquad\qquad \text{by item (4) of Definition 5.3.10} & \\
\quad = (\mu\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i\in I} \cup B \cup (B' \setminus B)) \setminus (B \setminus B') & \text{by (T.6)} \\
\quad = \mu\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i\in I} \cup B' & \text{by set theory}
\end{array}
$$

where $\uplus$ denotes disjoint union. In addition, from (1') and Lemma 5.4.4(1) we obtain $\Gamma'_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \Gamma'_3 \Vdash \mu'$, so we have:

$$
\begin{array}{ll}
(**)\ \mathsf{dom}(\mu') & = \mu'\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1, \Gamma'_2) \cup \{a_i\}_{i\in I}) \\
& = \mu'\text{-reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \uplus \mu'\text{-reach}(\mathsf{dom}(\Gamma'_2) \cup \{a_i\}_{i\in I})
\end{array}
$$

where the two equalities are respectively justified by items (3) and (4) of Definition 5.3.10. From (T.3), (T.4) we obtain μ-balanced$(\{a_i\}_{i\in I} \cup B)$, and then from (T.6) we get μ-balanced$(\mu$-reach$(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)))$. Therefore, by taking $\Gamma_I = \emptyset$ in (3') we obtain μ-reach$(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) = \mu'$-reach$(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1))$ and then from (*) and (**) we obtain (T.6'). We conclude this part of the proof with an application of rule [T-Running Transaction] to (1'), (T.2), (T.3), (T.4'), (T.5') and (T.6').

Regarding (2), we conclude from (2') and rule [WF-Run] of Figure 5.12. Regarding (3), let $\Gamma_J \subseteq \Gamma_R, [\Gamma_S]$ be such that μ-balanced$(\mu$-reach$(\mathsf{dom}(\Gamma_J, \Gamma)))$. Now take $\Gamma_I = \Gamma_J, [\Gamma_1]$ and observe that μ-balanced$(\mu$-reach$(\mathsf{dom}(\Gamma_I, \Gamma_3)))$. From (3') we conclude

$$\mu\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I)) = \mu'\text{-reach}(\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I))$$

which is (3) because $\mathsf{dom}(\Gamma_R, \Gamma_S, \Gamma_1) \setminus \mathsf{dom}(\Gamma_I) = \mathsf{dom}(\Gamma_R, \Gamma_S) \setminus \mathsf{dom}(\Gamma_J)$.

[R-Catch Exception] In this case $P = \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J}\texttt{throw}\ \mathsf{e}_k \mid P'' \rangle$ and $\mu = \mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2$ where $k \in J$ and $P' = Q_k$ and $\mu' = \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I}$. From rule [T-Running Transaction] we deduce:

- (L.1) $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (H.1) $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma_1, \{a_i : S_{ik}\}_{i\in I} \vdash P'$;
- (T.1) μ-balanced$(\{a_i : S_{ik}\}_{i\in I})$;
- (T.2) $\mathsf{local}(\Gamma_2)$;
- (T.3) $\{a_i\}_{i\in I} \cup \mathsf{dom}(\mu_2) = \mu$-reach$(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$.

Let $\Gamma'_0 = \Gamma_0 \setminus \mathsf{dom}(\mu_2)$ and $\Gamma' = \Gamma_1, \{a_i : S_{ik}\}_{i\in I}$. We only have to show that (H.2) $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and we prove the items of Definition 5.3.10. Items (1),

(2), and (4) are trivial because μ' has no more pointers than μ, some queues in μ have been emptied in μ', and duality of endpoint types associated with peer endpoints is preserved by (T.1). Regarding item (3), we have to show that $\mathsf{dom}(\mu') = \mathsf{dom}(\Gamma'_0, \Gamma_R, [\Gamma_S], \Gamma') = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma'))$. The first equality is easy. Regarding the second equality, we derive:

$$
\begin{array}{ll}
\mathsf{dom}(\mu) & \\
\quad = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \uplus \mathsf{dom}(\Gamma_R, \Gamma_1, \Gamma_2)) & \text{by item (3) of Definition 5.3.10} \\
\quad = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2)) & \\
\qquad\qquad \text{by item (4) of Definition 5.3.10} & \\
\quad = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, \Gamma_1)) \uplus \{a_i\}_{i\in I} \uplus \mathsf{dom}(\mu_2) & \text{by (T.3)} \\
\quad = \mathsf{dom}(\mu_1) \uplus \{a_i\}_{i\in I} \uplus \mathsf{dom}(\mu_2) & \text{by definition of } \mu
\end{array}
$$

where we write $\uplus$ for disjoint union. From the last equality we deduce

$$(*) \qquad \mathsf{dom}(\mu_1) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) = \mu_1\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1))$$

and now we have

$$
\begin{array}{lll}
\mathsf{dom}(\mu') = \mathsf{dom}(\mu_1) \uplus \{a_i\}_{i\in I} & & \text{by definition of } \mu' \\
\qquad = \mu_1\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma_1)) \cup \{a_i\}_{i\in I} & & \text{from (*)} \\
\qquad = \mu'\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma_R, [\Gamma_S], \Gamma')) & & \text{by definition of } \Gamma' \text{ and } \mu'
\end{array}
$$

We conclude (1) from (H.1), (H.2) and rule [T-Running Process]. Regarding (2), from (L.1) we know that ranks of all pointers a_i are preserved and from (T.2) that that the rank of all pointers that are no more in the environment is 0. Regarding (3), it holds trivially.

[R-Propagate Exception] Then $P = \langle \{a_i\}_{i\in I}, \mathsf{dom}(\mu_2), \{\mathsf{e}_j : Q_j\}_{j\in J} \texttt{throw}\ \mathsf{e} \mid P'' \rangle$ and $\mu = \mu_1, \{a_i \mapsto [b_i, \mathfrak{Q}_i]\}_{i\in I}, \mu_2$ and (E.1) $\mathsf{e}_j \neq \mathsf{e}$ for every $j \in J$ and $P' = \texttt{throw}\ \mathsf{e}$ and $\mu' = \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in I}$. From rule [T-Running Transaction] we deduce:

- (L.1) $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (T.1) $\tilde{\mathcal{E}}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [[\Gamma_S], \Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \fatsemi \texttt{throw}\ \mathsf{e} \mid P''$
- (T.2) $\mu\text{-}\mathsf{balanced}(\{a_i : S_{ij}\}_{i\in I})$;
- (T.3) $\mathsf{local}(\Gamma_2)$;
- (T.4) $\{a_i\}_{i\in I} \cup \mathsf{dom}(\mu_2) = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$.

From (T.1), [T-Running Parallel], [T-Running Process], and [T-Throw] we deduce $\mathsf{e} \in \tilde{\mathcal{E}}$. Let $\Gamma'_0 = \Gamma_0 \setminus \mathsf{dom}(\mu_2)$ and $\Gamma' = \Gamma_1, \{a_i : S_{ik}\}_{i\in I}$ for some arbitrary $k \in J$. By rule [T-Throw] we derive $\tilde{\mathcal{E}}; [\Gamma_S], \Gamma' \vdash \texttt{throw}\ \mathsf{e}$. The proof that $\Gamma'_0; \Gamma_R, [\Gamma_S], \Gamma' \Vdash \mu'$ and that items (2) and (3) hold is the same as for the case [R-Catch Exception]. □

The next two lemmas show further relationships between the free names of a process and the names occurring in the context used for typing it.

Lemma 5.4.8. *If* $\tilde{\emptyset};\Gamma \vdash P$, *then* $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P)$.

Proof. By induction on the derivation of $\tilde{\emptyset};\Gamma \vdash P$ and by cases on the last rule applied.

[T-Inaction] In this case $P = \texttt{done}$. From the hypotheses $\tilde{\emptyset};\Gamma \vdash \texttt{done}$ we conclude that $\Gamma = \emptyset$ and derive $\mathsf{dom}(\Gamma) = \emptyset = \mathsf{fn}(P)$.

[T-Invoke] In this case $\mathsf{dom}(\Gamma) = \tilde{u} = \mathsf{fn}(P)$.

[T-Open] In this case $P = \texttt{open}(a,b).Q$ and $\tilde{\emptyset};\Gamma, a : T, b : \overline{T} \vdash Q$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma, a : T, b : \overline{T}) \subseteq \mathsf{fn}(Q)$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma, a : T, b : \overline{T}) \setminus \{a, b\} \subseteq \mathsf{fn}(Q) \setminus \{a, b\} = \mathsf{fn}(P)$.

[T-Close] In this case $P = \texttt{close}(u).Q$, $\Gamma = \Gamma', u : \texttt{end}$ and $\tilde{\emptyset};\Gamma' \vdash Q$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma') \subseteq \mathsf{fn}(Q)$ and then we conclude $\mathsf{dom}(\Gamma) = \{u\} \cup \mathsf{dom}(\Gamma') \subseteq \{u\} \cup \mathsf{fn}(Q) = \mathsf{fn}(P)$.

[T-Send] In this case $P = u!\mathtt{m}_k(v).Q$, $\Gamma = \Gamma', u : \{!\mathtt{m}_i(S_i).T_i\}_{i\in I}, v : S$ and $\tilde{\emptyset};\Gamma', a : T_k \vdash P'$ for $k \in I$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma', u : T_k) \subseteq \mathsf{fn}(Q)$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u\} \cup \{v\} \subseteq \mathsf{fn}(Q) \cup \{v\} \subseteq \mathsf{fn}(P)$.

[T-Receive] In this case $P = \sum_{i\in I\cup J} u?\mathtt{m}_i(x_i).P_i$, $\Gamma = \Gamma', u : \{?\mathtt{m}_i(S_i).T_i\}_{i\in I}$ and for all $i \in I$ we have $\tilde{\emptyset};\Gamma', u : T_i, x_i : S_i' \vdash P_i$ where $S_i \leqslant S_i'$. By induction hypothesis for all $i \in I$ we obtain $\mathsf{dom}(\Gamma', u : T_i, x_i : S_i') \subseteq \mathsf{fn}(P_i)$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u\} = \bigcup_{i\in I}(\mathsf{dom}(\Gamma', x_i : S_i') \setminus \{x_i\}) \cup \{u\} \subseteq \bigcup_{i\in I}(\mathsf{fn}(P_i) \setminus \{x_i\}) \cup \{u\} \subseteq \bigcup_{i\in I\cup J}(\mathsf{fn}(P_i) \setminus \{x_i\}) \cup \{u\} = \mathsf{fn}(P)$.

[T-Choice] In this case $P = P_1 \oplus P_2$ and $\tilde{\emptyset};\Gamma \vdash P_i$ for $i \in \{1, 2\}$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P_i)$ for $i \in \{1, 2\}$ and then we conclude $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) = \mathsf{fn}(P)$.

[T-Parallel] In this case $P = P_1 \mid P_2$, $\Gamma = \Gamma_1, \Gamma_2$ and $\tilde{\emptyset};\Gamma_i \vdash P_i$ for $i \in \{1, 2\}$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma_i) \subseteq \mathsf{fn}(P_i)$ for $i \in \{1, 2\}$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \subseteq \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) = \mathsf{fn}(P)$.

[T-Try] In this case $P = \texttt{try}(\{u_i\}_{i\in I})\ \{\mathsf{e}_j : R_j\}_{j\in J} Q$, $\Gamma = \Gamma', \{u_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J}[\![T_i]\!]_{i\in I}$, $\tilde{\emptyset}\{\mathsf{e}_j\}_{j\in J}; [\Gamma'], \{u_i : T_i\}_{i\in I} \vdash Q$ and $\tilde{\emptyset};\Gamma', \{u_i : S_{ij}\}_{i\in I} \vdash R_j$ for every $j \in J$. We distinguish two subcases, according to whether J is empty or not. If $J = \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}([\Gamma'], \{u_i : T_i\}_{i\in I}) \subseteq \mathsf{fn}(Q)$ and then we conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I} \subseteq \mathsf{fn}(Q) = \mathsf{fn}(P)$. If $J \neq \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}(\Gamma', \{u_i : S_{ij}\}_{i\in I}) \subseteq \mathsf{fn}(R_j)$ for every $j \in J$. We conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \cup \{u_i\}_{i\in I} \subseteq \bigcup_{j\in J} \mathsf{fn}(R_j) \subseteq \mathsf{fn}(P)$.

[T-Throw] This case is impossible because the exception environment consists of a sequence of empty sets of exceptions, hence all the exceptions thrown in P are caught.

[T-Commit] In this case $\Gamma = [\Gamma_1], \{u_i : [\![T_i]\!]_{i\in I}, \Gamma_2$, $P = \texttt{commit}(\{u_i\}_{i\in I}).Q$ and $\emptyset;\Gamma_1, \{u_i : T_i\}_{i\in I}, \Gamma_2 \vdash Q$. By induction hypothesis we obtain $\mathsf{dom}(\Gamma_1) \cup \bigcup_{i\in I}\{u_i\} \cup$

$\mathsf{dom}(\Gamma_2) \subseteq \mathsf{fn}(Q)$. We conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma_1) \cup \bigcup_{i\in I}\{u_i\} \cup \mathsf{dom}(\Gamma_2) \subseteq \mathsf{fn}(Q) \subseteq \mathsf{fn}(P)$. □

Lemma 5.4.9. *If* $\tilde{\emptyset}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P$, *then* $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(P)$.

Proof. By induction on the derivation of $\tilde{\emptyset}; \Gamma_0; \Gamma_R; \Gamma \vdash \mu \wr P$ and by cases on the last rule applied. Cases [T-Running Process] and [T-Running Parallel] are easily solved by Lemma 5.4.8 and the induction hypothesis, respectively. Regarding [T-Running Transaction], we have:

- $P = \langle \{a_i\}_{i\in I}, B, \{\mathsf{e}_j : R_j\}_{j\in J} Q\rangle$;
- $\Gamma = \Gamma_1, \{a_i : \{\mathsf{e}_j : S_{ij}\}_{j\in J} T_i\}_{i\in I}, \Gamma_2$;
- (*) $\{a_i\}_{i\in I} \cup B = \mu\text{-}\mathsf{reach}(\{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2))$;
- $\tilde{\emptyset}\{\mathsf{e}_j\}_{j\in J}; \Gamma_0; \Gamma_R; [\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2 \vdash \mu \wr Q$;
- $\tilde{\emptyset}; \Gamma_1, \{a_i : S_{ij}\}_{i\in I} \vdash R_j$ for all $j \in J$.

We distinguish two cases, according to whether J is empty or not. If $J = \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}([\Gamma_1], \{a_i : T_i\}_{i\in I}, \Gamma_2) \subseteq \mathsf{fn}(Q)$ and we conclude $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(Q) \subseteq \mathsf{fn}(P)$. If $J \neq \emptyset$, then by induction hypothesis we obtain $\mathsf{dom}(\Gamma_1, \{a_i : S_{ij}\}_{i\in I}) \subseteq \mathsf{fn}(R_j)$ for every $j \in J$. From (*) we deduce $\mathsf{dom}(\Gamma_2) \subseteq \{a_i\}_{i\in I} \cup B$. We conclude $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma_1) \cup \{a_i\}_{i\in I} \cup \mathsf{dom}(\Gamma_2) \subseteq \bigcup_{j\in J} \mathsf{fn}(R_j) \cup \{a_i\}_{i\in I} \cup B \subseteq \mathsf{fn}(P)$. □

We can now prove the soundness of the type system.

Theorem 5.4.10 (Type safety). *Let* $\vdash P$. *Then* P *is well behaved.*

Proof. From the hypothesis $\vdash P$ we deduce $\tilde{\emptyset}; \emptyset; \emptyset; \emptyset \vdash \emptyset \wr P$. Consider an arbitrary derivation $\emptyset \wr P \to^* \mu \wr Q$. From Theorem 5.4.7 we deduce that there exist Γ_0 and Γ such that $\tilde{\emptyset}; \Gamma_0; \emptyset; \Gamma \vdash \mu \wr Q$ and, from Lemma 5.4.4, we obtain $\Gamma_0; \Gamma \Vdash \mu$.

Regarding condition (1) of Definition 5.2.6, from $\Gamma_0; \Gamma \Vdash \mu$ and Definition 5.3.10 we know $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.

By Lemma 5.4.9, we have $\mathsf{dom}(\Gamma) \subseteq \mathsf{fn}(Q)$ and then, because μ-reach is monotone, we obtain $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma)) \subseteq \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$.

Regarding condition (2) of Definition 5.2.6, suppose that $Q \equiv Q_1 \mid Q_2$ and $\mu \wr Q_1 \not\to$ and $Q_1 \not\equiv \texttt{throw}\ \mathsf{e} \mid Q_1'$ (the last hypothesis being granted by the fact that Q is well typed in the $\tilde{\emptyset}$ exception environment). We prove $\mu \wr Q_1 \downarrow$ by induction on Q_1.

- ($Q_1 = \texttt{done}$) We conclude with an application of rule [ST-Inactive].
- ($Q_1 = \texttt{open}(a, b).R$) Because we have assumed that there are infinitely many pointers and structural congruence includes alpha renaming, we may assume $a, b \notin \mathsf{dom}(\mu)$. Then $\mu \wr Q_1 \to$, which contradicts the hypothesis, therefore this case is impossible.

- $(Q_1 = \texttt{close}(a).R)$ From [T-Running Parallel], [T-Running Process], [T-Close] and item (3) of Definition 5.3.10 we obtain $a \in \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma_0, \Gamma) = \mathsf{dom}(\mu)$, and now $\mu \fatsemi Q_1 \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.

- $(Q_1 = R_1 \oplus R_2)$ This case is impossible because $\mu \fatsemi R_1 \oplus R_2$ always reduces.

- $(Q_1 = a!\mathtt{m}(c).R)$ From rules [T-Running Process], [T-Running Parallel] and [T-Send] we obtain $\Gamma \vdash a : T$ where T is an internal choice and then from item (3) of Definition 5.3.10 we have $a \in \mathsf{dom}(\mu)$. From item (2) of Definition 5.3.10 we deduce that the queue associated with a is empty and also that the peer of a, say b, is still allocated in μ for otherwise T would have to be $\texttt{end}$. Then $\mu \fatsemi Q_1 \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.

- $(Q_1 = \sum_{i \in I} a?\mathtt{m}_i(x_i).R_i)$ Then $a \mapsto [b, \mathfrak{Q}] \in \mu$ and the messages and arguments in $\mathfrak{Q}$ are consistent with the type of endpoint a. The only case when $\mu \fatsemi \sum_{i \in I} a?\mathtt{m}_i(x_i).R_i$ does not reduce is when $\mathfrak{Q} = \varepsilon$, therefore we conclude $\mu \fatsemi Q_1 \downarrow$ by an application of rule [ST-Input].

- $(Q_1 = R_1 \mid R_2)$ From the hypothesis $\mu \fatsemi Q_1 \not\rightarrow$ we deduce $\mu \fatsemi R_i \not\rightarrow$ for $i \in \{1, 2\}$. From the hypothesis $Q_1 \not\equiv \texttt{throw}\ \mathsf{e} \mid Q_1'$ we deduce $R_i \not\equiv \texttt{throw}\ \mathsf{e} \mid R_i'$ for $i \in \{1, 2\}$. By induction hypothesis we obtain $\mu \fatsemi R_i \downarrow$ for $i \in \{1, 2\}$. We conclude with an application of rule [ST-Parallel].

- $(Q_1 = \texttt{try}(A)\ \{\mathsf{e}_j : R_j\}_{j \in J} R')$ From the hypothesis $\mu \fatsemi Q_1 \not\rightarrow$ we deduce $\neg \mu\text{-}\mathsf{balanced}(A)$. We conclude with an application of rule [ST-Try].

- $(Q_1 = \texttt{throw}\ \mathsf{e})$ This case is impossible by hypothesis.

- $(Q_1 = \texttt{commit}(A).R)$ We conclude immediately with an application of rule [ST-Commit].

- $(Q_1 = X\langle \tilde{a} \rangle)$ From rule [T-Invoke] we deduce $X(\tilde{u}) \stackrel{\text{def}}{=} R$ is a definition and $\tilde{a}$ and $\tilde{u}$ have the same length. Then $\mu \fatsemi X\langle \tilde{a} \rangle \rightarrow$ which contradicts the hypothesis, therefore this case is impossible.

- $(Q_1 = \langle A, B, \{\mathsf{e}_j : R_j\}_{j \in J} R' \rangle)$ From the hypothesis $\mu \fatsemi Q_1 \not\rightarrow$ we deduce $\mu \fatsemi R' \not\rightarrow$ and $R' \not\equiv \prod_{i \in I} \texttt{commit}(A_i).R_i$ and $R' \not\equiv \texttt{throw}\ \mathsf{e} | R''$ since these are the cases when $\mu \fatsemi Q_1$ does reduce. By induction hypothesis we deduce $\mu \fatsemi R' \downarrow$ and we conclude with an application of rule [ST-Running Transaction].

□

We conclude this section with two examples showing how transaction types increase the expressiveness of session types by allowing the safe modeling of timeouts and mixed choices.

Example 5.4.11 (timeouts). In some cases it is possible and desirable to establish a timeout for a receive operation to succeed. Using transaction types and exceptions, it is easy to model timeouts in our process language. As an example, suppose one is interested in modeling a process

$$a?\mathtt{m}(x).P + \tau.Q$$

which behaves as P as soon as it receives an $\mathtt{m}$ message from endpoint a and reduces to Q through an internal τ move if no message is received after some unspecified amount of time. This can be modeled by means of the process

$$\begin{array}{l}\mathtt{try}(a)\ \{\mathtt{timeout} : Q\} \\ \qquad a?\mathtt{m}(x).\mathtt{commit}(a).P \\ \quad \mid\ \mathtt{try}(\{\})\ \{\mathtt{ok} : \mathtt{done}\}(\mathtt{throw}\ \mathtt{ok} \oplus \mathtt{throw}\ \mathtt{timeout})\end{array}$$

where the nondeterministic choice between throwing `ok` or `timeout` is the abstract representation of the mechanism that activates the timeout. Note that the modeling uses exception propagation to trigger the timeout. The type associated with a is $\{\mathtt{timeout} : S\}[\![?\mathtt{m}(T).]\!]T$ where T describes the behavior of P on a if the $\mathtt{m}$ message is received within the timeout, while S describes the behavior of Q on a if the timeout expires. Note that the modeling in this example is not meant to suggest an actual implementation of the timeout mechanism, but rather to describe its effect in abstract terms. In particular, the $\mathtt{m}$ message may have already been sent by the time the timeout expires (and the exception `timeout` is thrown). Yet, the sender of the message should be aware of this eventuality, it should be notified in case the timeout expires, and it should provide a suitable recovery action for this eventuality, possibly involving the resending of the $\mathtt{m}$ message. This is all guaranteed by the fact that the sender, which uses the peer endpoint of a, must do so according to the endpoint type $\{\mathtt{timeout} : \overline{S}\}[\![!\mathtt{m}(T).]\!]\overline{T}$. △

Example 5.4.12 (mixed choices). The Sing$^\#$ implementation allows the definition of contracts with so-called *mixed choices*, namely states in which there are two (or more) alternative operations involving both inputs and outputs. If mixed choices were allowed in our type language we would have, for example, endpoint types of the form $\{!\mathtt{a}(t).T, ?\mathtt{b}(s).S\}$ allowing either sending an `a` message or receiving a `b` message. Mixed choices break the half-duplex communication modality and are known to make protocols less robust and prone to deadlock [136, 146]. Yet, they can be safely modeled using transaction types as, for example

$$T_m = \{\mathtt{a} : !\mathtt{a}(T).T, \mathtt{b} : ?\mathtt{b}(s).S\}[\![\ ]\!]\mathtt{end}$$

The intuition is that a process behaving as T_m may throw either an `a` or a `b` exception to notify its party as to which operation (output an `a` message or input a `b` message) it will perform. An example of such process is

$$\mathtt{try}(c)\{\mathtt{a} : c!\mathtt{a}(a).P, \mathtt{b} : c?\mathtt{b}x.Q\}(\mathtt{commit}(c).\mathtt{close}(c) \mid X\langle a\rangle)$$

where

$$X(y) \stackrel{\text{def}}{=} X\langle y\rangle \oplus \mathtt{throw}\ \mathtt{a}$$

represents an internal computation that may eventually throw the a exception and cause the sending of the a message.

Below is part of the proof derivation showing that the process is well typed in the environment $c : T_m, u : T$. The branch related to the b exception has been omitted, but it is analogous to the one for the a exception; where necessary, the exception environment $\mathcal{E} = \{\mathtt{a}, \mathtt{b}\}$ is used:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E};\emptyset \vdash \mathtt{done}}{\mathcal{E}; c : \mathtt{end} \vdash \mathtt{close}(c)}}{\mathcal{E}; c : \rrbracket\mathtt{end} \vdash \mathtt{commit}(c).\mathtt{close}(c)} \quad \mathcal{E}; u : [T] \vdash X\langle a\rangle}{\mathcal{E}; c : \rrbracket\mathtt{end}, u : [T] \vdash \mathtt{commit}(c).\mathtt{close}(c) \mid X\langle a\rangle} \quad \dfrac{\dfrac{\vdots}{c : T \vdash P}}{c : !\mathtt{a}(T).T, u : T \vdash c!\mathtt{a}(a).P} \; \vdots}{c : T_m, u : T \vdash \mathtt{try}(c)\{\mathtt{a} : c!\mathtt{a}(a).P, \mathtt{b} : c?\mathtt{b}(x).Q\}(\mathtt{commit}(c).\mathtt{close}(c) \mid X\langle a\rangle)}$$

Finally,

$$\dfrac{\mathcal{E}; u : [T] \vdash X\langle u\rangle \quad \mathcal{E}; u : [T] \vdash \mathtt{throw}\ \mathtt{a}}{\mathcal{E}; u : [T] \vdash X\langle u\rangle \oplus \mathtt{throw}\ \mathtt{a}}$$

is the proof tree proving that the definition of $X\langle u\rangle$ is well typed. △

5.5 Conclusion and related work

We have formalized a core language of processes that communicate and synchronize through the copyless message passing paradigm and can throw exceptions. In this context, where the sharing of data and explicit memory allocation require controlled policies on the ownership of heap-allocated objects, special care must be taken when exceptions are thrown to prevent communication errors (arising from misaligned states of channel endpoints) and memory leaks (resulting from messages forgotten in endpoint queues). We have studied a type system guaranteeing some safety properties, in particular that well-typed processes are free from communication errors and do no leak memory even in presence of (caught) exceptions. We have taken advantage of invariants guaranteed by the type system for taming the implementation costs of exception handling: the queues of endpoints involved in a transaction are guaranteed to be empty when the transaction starts, so that state restoration in case of exception simply means emptying such queues; also, only endpoints local to a transaction can be freed inside the transaction, so that state restoration in case of exception does not involve re-allocations.

The choice of Sing# as our reference language has been motivated by the fact that the Singularity code base provides concrete programming patterns that the formal model is supposed to cover (for example, previous works on exception handling for session-oriented languages by [25, 23] do not consider delegation inside try-blocks, which instead is ubiquitous in Sing#). In addition, Sing# already accommodates channel contracts, which play a crucial role in our formalization. However, we claim that our approach is abstract enough to be applicable to other

programming languages and paradigms, provided that suitable type information (possibly in the form of code annotations) is attached to channel endpoints.

To measure the practical impact of our mechanism of exception handling, one can take advantage of the availability of Singularity's source code for verifying whether the constraints imposed by the type discipline are reasonable in practice. Even without thorough investigations, however, we are able to provide some favorable arguments to our type discipline, in particular with respect to the weight and rank restrictions on types. As regards type weights, one has to consider that messages allocated on the exchange heap are explicitly managed by means of reference counting [83], which notoriously falls short in handling cyclic data structures, and that the finite-weight restriction on the type of communicated messages is just aimed at preventing cycles in the exchange heap. Regarding ranks, the subclass of well-formed, null-ranked types are just those in which transactions are properly balanced. In fact, the notion of well formedness arises solely because of our choice of modeling transactions using two matching constructs `try` and `commit` marking their beginning and end. Their balancing arises naturally in a structured language such as Sing#.

Even if the type system has been tailored for a process calculus with a minimal set of critical features, it can be easily extended to incorporate commonly used programming constructs. Some restrictions of the type system can also be relaxed without endangering its soundness. For example, according to rule [T-Send], only local endpoints (those that have an unsealed and null-ranked type) can be sent as messages inside transactions. This restriction results from the syntax of endpoint types (requiring that message arguments must have an unsealed type) and from rule [WF-Prefix] regarding well-formed endpoint types (requiring that message argument types must have null rank). Endpoints with a sealed type cannot be accessed from within a transaction, because all the operations that modify the state of the heap require the access to endpoints with unsealed type. Therefore, we claim that endpoints with a sealed type are also safe to be sent as messages. In case an exception is thrown, the only thing that must be restored is their ownership at the beginning of the transaction. Nonetheless, the proof of this relaxed discipline seems to require a non-trivial modification of rule [T-Running Transaction], which is already quite elaborate in the present state, to account for the fact that the state of endpoints with a sealed type can change as the result of concurrent threads that execute *outside* of the transaction.

The research presented in this chapter fits in the broad spectrum of works developing hybrid static/dynamic techniques for the controlled management of the heap, and relies on invariants regarding the configuration of heap-allocated structures for enabling the efficient implementation of transactional mechanisms. Controlled heap management has fostered the development of a wide spectrum of techniques aimed at the most diverse purposes, of which we provide here a small account.

Pure functional programming languages are excellent candidates for the implementation of implicitly parallel computations. Parallelism is usually achieved by allowing multiple processing units to independently reduce disjoint parts of a program (represented as a graph) stored in a (possibly virtual) shared memory.

Both hardware [117] and software [143] architectures have been explored. In these architectures, the crucial aspect is to achieve an optimal distribution of tasks, taking into account the fact that each processing unit often has its own local memory where subgraphs to be reduced must be copied, and that for improved efficiency it is necessary to take into proper account locality properties of the program [99].

Type-based approaches for enforcing and reasoning on properties of the heap are also popular. The seminal work by [142] describes an *effect type system* for region-based memory management that allows for efficient allocation and deallocation of related heap structures. Interestingly, effects can be seen as primitive forms of behavioral types. There exist type systems with resource annotations that allow computing bounds on the heap space usage of functions and methods. Examples of such type systems are given in [76] for (first-order) functional programs and in [77] for object-oriented programs. These approaches are usually motivated by the need to provide firm guarantees in code meant to be executed in embedded systems with constrained resources. Such forms of analyses can be carried out also for untyped programs. For instance, [5] works directly on Java bytecode, while [65] studies bounded time/space semantics for a functional and concurrent language.

The paper [137] studies a type discipline for safe resource deallocation in concurrent programs with shared memory. The idea is to associate each shared resource with a *fractional ownership*, namely a rational number in the interval $[0, 1]$, which denotes the level of sharing of the resource: 0 means that the resource is not owned, 1 means that the resource is owned exclusively, while any intermediate rational number indicates a shared ownership of the resource (often constraining the operations allowed on it). The type system allows a thread to deallocate a resource only if the thread is the only owner of the resource *and* if all of the other resources contained in it have been deallocated or transferred to other threads. In our type system, a similar effect is achieved by the combination of the type rule for the `close` primitive (which allows for the deallocation of endpoints only when they are owned exclusively) and the notion of well-typed heap (which guarantees endpoints with type `end` to have an empty queue).

There are strong analogies between our endpoint types and *usage expressions* defined by [86], which are used for controlling access to resources (such as files and memory) in a functional language with exceptions. Usage expressions are akin to behavioral types and describe the valid sequence of operations allowed on some resource. In particular, [86] defines a usage constructor $U_1;_E U_2$ where U_1 describes how the resource is accessed under "nominal" conditions while U_2 describes how the resource is accessed if an exception is thrown. The structure of this usage constructor resembles that of a transaction type $\{E : U_2\}[\![U_1$. Because usage expressions can be composed in sequence, there is no need to explicitly mark the points where the scope of an exception ends.

The present work continues the type-based formalization of Singularity OS described by [17]. To simplify the formal development of the present paper we dropped polymorphism and non-linear types from the type system in [17]. These are orthogonal features that are independent of exception handling and can be added without affecting the results we have presented here. A radically different

approach for the static analysis of Singularity processes is explored by [146, 147], where the authors develop a proof system based on a variant of *separation logic*. Exceptions are not taken into account in these works.

The works more closely related to ours, and which we used as starting points, are by [25] and [23]. In [25], which was the first to investigate exceptions in calculi for session-oriented interactions and to propose type constructs to describe explicitly, at the type level, the handling of exceptional events, it is possible to associate an exception handler to a whole (dyadic) session; [23] generalizes this idea to multiparty sessions (those with multiple participants) and allows the same channel to be involved, at different times, in different `try` blocks, each with its own dedicated exception handler. In both [25] and [23] it is possible that messages already present in channel queues at the time an exception occurs are forgotten. In our context, this would immediately yield memory leaks, which we avoid by keeping track of the resources allocated during a transaction and by restoring the system to a consistent configuration in case an exception is thrown. State restoration is made possible in our context because the system is not distributed and the heap is shared by the communicating processes. Neither [25] nor [23] consider session delegation, namely the communication of channels. Also, in [23] the type system forces inner `try` blocks to use a subset of the channels involved in outer blocks. We relax these restrictions and allow locally created channels to be involved in inner transactions. The most notable difference between [23] and the present work regards the semantics of exceptions in nested transactions: in [23], an exception thrown in one transaction is suspended as long as there are active handlers in the nested ones. This semantics is motivated by the observation that, in a distributed setting, it may be desirable to complete the execution of potentially critical handlers before outermost handlers take control. Our semantics allows handlers of outer transactions to take control at any time following the throwing of an exception. As a consequence, more constrained policies, such as the one adopted in [23], can be implemented without invalidating the results presented in our work.

The recent interest on Web services has spawned a number of works investigating (long running) transactions in a distributed setting; a detailed survey with many references is provided by [49]. In our context, the components Q_i of a process $\langle A, B, \{\mathsf{e}_i : Q_i\}_{i\in I} P\rangle$ are analogous to *compensation handlers*. The main difference between our handlers and those used for compensations is that, in the latter case, it is usually made the assumption that it is not possible to restore the state of the system as it was at the beginning of the transaction. In our case, state restoration is made possible by the fact that the system is local and all the interactions occur through shared memory. In this context, we can rely on some native support from the runtime system to properly cleanup the state of the system and avoid memory leaks.

The operational semantics of exceptions and exception handling in the present paper has been loosely inspired by that of Haskell memory transactions described by [67]. In particular, our semantics describes *what* happens when an exception is thrown but not *how* exception notification and state restoration are implemented. In this sense our semantics is somewhat more abstract than the semantics given

in similar works [23]. The semantics of [67] uses a clever combination of small- and big-step reduction rules and is even more abstract than ours. Technically, having an abstract semantics is an advantage because it allows the meaning of programs to be expressed more concisely and in an implementation-independent way. However, because the big-step semantics is affected by diverging programs, it is more appropriate in a functional setting where non-termination is typically considered as a misbehavior. In our context, where non-terminating processes are useful, we had to resort to a more detailed operational semantics that dynamically keeps track of the allocated memory within a transaction.

[43] puts forward a programming abstraction called *transactional events* for the modular composition of communication events into transactions with an all-or-nothing semantics. Their approach focuses on finding synchronization paths between threads communicating synchronously, while in our case transactions are required for preserving type consistency of endpoints and for undoing the effects of asynchronous communication.

Inadequacy of the standard error handling mechanisms provided by mainstream programming languages has already been recognized, even in sequential and communication-free scenarios. [149, 148, 150] develop a static analysis technique that spots error handling mistakes concerning proper resource release. Their technique is based on finite-state automata (in other words, a basic form of behavioral type) for keeping track of the state of resources along all possible execution paths. They also propose a more effective mechanism for preventing runtime errors. The basic idea is to accumulate compensation actions regarding resources on a *compensation stack* as resources are allocated. This technique closely resembles dynamic compensations in [49]. Because of their dynamic nature, compensation stacks do not provide any assistance as far as type consistency is concerned.

Chapter 6

Conclusion

Concurrency and distribution are big challenges for verification because of the presence of shared data and nondeterministically interleaved computations, and the lack of centralized control and mutual trust. In Chapters 3 and 4, we verify security and privacy properties of distributed networks with data in XML and RDF format. Process languages are inspired by existing query languages, featuring commands for reading, writing and changing data. In Chapter 3 role-based access is used to control access to the data, while in Chapter 4 access is controlled by privacy protection polices of the users. We define relevant security and privacy properties in terms of access rights. We use static type systems to show that networks are correct with respect to these properties and that security and privacy policies of locations and users are respected during computation. In Chapter 5 we investigate the prevention of memory errors and leaks as well as the communication errors in copyless messaging communication paradigm. To formalize the semantics of processes we draw inspiration from software transactional memories: in our case a transaction is a process that is meant to accomplish some exchange of messages and that should either be executed completely, or should have no observable effect if aborted by an exception. In this chapter we study a type discipline for copyless messaging that, together with some minimal support from the runtime system, is able to guarantee the absence of communication errors, memory faults, and memory leaks in the presence of exceptions. We have taken advantage of invariants guaranteed by the type system for taming the implementation costs of exception handling: the queues of endpoints involved in a transaction are guaranteed to be empty when the transaction starts, so that state restoration in case of exception simply means emptying such queues; also, only endpoints local to a transaction can be freed inside the transaction, so that state restoration in case of exception does not involve re-allocations.

There is a fundamental difference between the models in Chapters 3 and 4, and the model in Section 5. The first two models deal with distributed systems, while the third one is designed for a local setting. The soundness results hold as long all the terms are well typed. This is relatively easy to ensure in a local setting, while in a distributed one it is not. Situations in which it is not known if all the components of a network have been type checked require further restrictions of the presented typing disciplines, as well as additional static and dynamic checks, based on, for example, trustworthiness of the components.

Some terms that are actually safe may be rejected by a type system due to its impreciseness. By introducing a subtyping relation it is possible to type more terms. In Chapter 5 we show that the exception annotations within types induce an original subtyping relation that sheds light on the differences between exceptions and regular messages. Type systems of Chapter 3 can be extended with subtyping relation, as suggested in [87]. After completing the material for the thesis, I have been working on the preciseness of subtyping relation for multiparty session types [39].

My current work includes a typed model for specifying communication and dynamic authorization handling [60]. We exploited this idea in multiparty conversations setting and combined it with roles [61]. We are working on refining the notion of delegating authorizations with delegating usages and sending them separately from the content.

Besides defining which participants can communicate and in which direction, there might be the need to constrain or examine the content of messages exchanged. Such policies are very expressive but difficult to analyse due to their very essence, content-dependence. This scenario is of interest in avionics industry and has already been analysed in [100]. In that work, the syntax of a fragment of C programming language is augmented with content-dependant labels and programs are verified using a tool based on the proposed type system. Concrete unresolved issues of [100] for the verification tool and the type system presented there (support of functions, policy polymorphism and inference, extension of the observed fragment of the language, push of the policy specification outside the code) are related to the domain of my research. A more abstract model, with communication primitives and authorities on whose behalf programs are running, could be a good base for reasoning about properties of systems in presence of content-based policies. Authorities/participants in communication can be further abstracted as channels, so that each participant is represented by a unique channel and a form of duality of policies could be used to type channels.

Bibliography

[1] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387 – 415, 2003.

[2] M. Abadi, R. Corin, and C. Fournet. Computational secrecy by typing for the pi calculus. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems*, volume 4279 of *LNCS*, pages 253–269. Springer, 2006.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47. ACM, 1997.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proceedings of the 6th International Symposium on Memory Management (ISMM'07)*, pages 105–116. ACM, 2007.

[6] ANSI. Information technology - role based access control. INCITS 359-2012, 2012.

[7] G. Bateson. *Steps to an Ecology of Mind: Collected Essays in Anthropology, Psychiatry, Evolution, and Epistemology*. University of Chicago Press, 1972.

[8] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. RDF 1.1 Turtle. Available at http://www.w3.org/TR/turtle/, 2014.

[9] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0 (second edition). Available at https://www.w3.org/TR/xpath20/, 2010.

[10] A. Berglund, M. F. A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM) (second edition). Available at https://www.w3.org/TR/xpath-datamodel/, 2010.

[11] T. Berners-Lee. Linked data - design issues. Available at http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[12] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[13] Y. Bertot and P. Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer, 2004.

[14] C. Bizer. The emerging web of linked data. *IEEE Intelligent Systems*, 24(5):87–92, 2009.

[15] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[16] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language (second edition). Available at https://www.w3.org/TR/xquery/, 2010.

[17] V. Bono and L. Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8:1–50, 2012.

[18] C. Braghin, D. Gorla, and V. Sassone. Role-based access control for a distributed calculus. *Journal of Computer Security*, 14(2):113–155, 2006.

[19] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth edition). Available at https://www.w3.org/TR/xml/, 2008.

[20] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[21] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication and mobility control in boxed ambients. *Information and Computation*, 202(1):39–86, 2005.

[22] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, FirstView:1–43, 2015.

[23] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. In *Proceedings of IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, (FSTTCS'10)*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[24] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26:156–205, 2016.

[25] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08)*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[26] L. Cardelli. Brane calculi. In *Revised Selected Papers of International Conference on Computational Methods in Systems Biology (CMSB'04)*, volume 3082 of *LNCS*, pages 257–278. Springer, 2004.

[27] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the Ambient Calculus. *Information and Computation*, 177(2):160–194, 2002.

[28] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'98)*. ACM, 1998.

[29] G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal calculus. *Information and Computation*, 201(1):1–54, 2005.

[30] A. Church. A formulation of the simple theory of types. *Jurnal of Symbolic Logic*, 5(2):56–68, 1940.

[31] G. Ciobanu, R. Horne, and V. Sassone. Minimal type inference for Linked Data consumers. *Journal of Logical and Algebraic Methods in Programming*, 84(4):485–504, 2015.

[32] G. Ciobanu, R. Horne, and V. Sassone. A descriptive type foundation for RDF schema. *Journal of Logical and Algebraic Methods in Programming*, 2016. In press.

[33] A. Compagnoni, E. Gunter, and P. Bidinger. Role-based access control for boxed ambients. *Theoretical Computer Science*, 398(1-3):203–216, 2008.

[34] R. L. Constable, S. F. Allen, M. Bromley, R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[35] M. Coppo, M. Dezani-Ciancaglini, and E. Giovannetti. Types for ambient and process mobility. *Mathematical Structures in Computer Science*, 18:221–290, 2008.

[36] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. RDF 1.1 concepts and abstract syntax. Available at http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014.

[37] R. de Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[38] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. In *Proceedings of the Proceedings of the 6th International Conference on Web Services and Formal Methods (WS-FM'09)*, volume 6194 of *LNCS*, pages 1–28. Springer, 2009.

[39] M. Dezani-Ciancaglini, S. Ghilezan, S. Jakšić, J. Pantović, and N. Yoshida. Precise subtyping for synchronous multiparty sessions. In *Proceedings of Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2015)*, volume 203 of *EPTCS*, pages 15–26. Open Publishing Association, 2015.

[40] M. Dezani-Ciancaglini, S. Ghilezan, S. Jakšić, and J. Pantović. Types for role-based access control of dynamic web data. In *Proceedings of the 19th International Workshop on Functional and Constraint Logic Programming*, volume 6559 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 2011.

[41] M. Dezani-Ciancaglini, S. Ghilezan, J. Pantović, and D. Varacca. Security types for dynamic web data. *Theoretical Computer Science*, 402(2-3):156–171, 2008.

[42] M. Dezani-Ciancaglini, R. Horne, and V. Sassone. Tracing where and who provenance in Linked Data: A calculus. *Theoretical Computer Science*, 464:113–129, 2012.

[43] K. Donnelly and M. Fluet. Transactional Events. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 124–135. ACM, 2006.

[44] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st EuroSys Conference (EuroSys'06)*, pages 177–190. ACM, 2006.

[45] S. Fernández, F. Giasson, and K. Idehen. SIOC ontology: Applications and implementation status. Available at http://rdfs.org/sioc/applications/#creating-exporters/, 2009.

[46] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, 1999.

[47] D. F. Ferraiolo, D. R. Kuhn, and R. S. Sandhu. Rôle-based access control. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, pages 554–563. NIST, 1992.

[48] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[49] C. Ferreira, I. Lanese, A. Ravara, H. T. Vieira, and G. Zavattaro. Advanced mechanisms for service combination and transactions. In *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*, pages 302–325. Springer, 2011.

[50] S. Finley. Ext2 on Singularity. Technical report, University of Wisconsin, 2008. Available at http://pages.cs.wisc.edu/~remzi/Classes/736/Spring2008/Projects/Scott/Ext2%20On%20Singularity.pdf.

[51] C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM'00*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.

[52] C. Fournet, A. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 31–48. IEEE, 2007.

[53] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *The Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.

[54] F. Gandon, G. Schreiber, and D. Beckett. RDF 1.1 XML syntax. Available at http://www.w3.org/TR/rdf-syntax-grammar/, 2014.

[55] P. Gardner and S. Maffeis. Modelling dynamic web data. *Theoretical Computer Science*, 342(1):104–131, 2005.

[56] P. Garralda, E. Bonelli, A. Compagnoni, and M. Dezani-Ciancaglini. Boxed ambients with communication interfaces. *Mathematical Structures in Computer Science*, 17:1–59, 2007.

[57] S. Gay and M. Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[58] S. J. Gay. A sort inference algorithm for the polyadic &pgr;-calculus. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 429–438. ACM, 1993.

[59] P. Gearon, A. Passant, and A. Polleres. SPARQL 1.1 update. Available at http://www.w3.org/TR/2013/REC-sparql11-update-20130321/, 2013.

[60] S. Ghilezan, S. Jakšić, J. Pantović, J. A. Pérez, and H. T. Vieira. A typed model for dynamic authorizations. In *Proceedings of Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2015)*, volume 203 of *EPTCS*, pages 27–36. Open Publishing Association, 2015.

[61] S. Ghilezan, S. Jakšić, J. Pantović, J. A. Pérez, and H. T. Vieira. Dynamic role authorization in multiparty conversations. *Formal Aspects of Computing*, 28(4):643–667, 2016.

[62] M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In *Proceedings of the 21th International Conference on*

Concurrency Theory (CONCUR'10), volume 6269 of *LNCS*, pages 432–446, 2010.

[63] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. *Logical Methods in Computer Science*, 1(3):331–353, 2005.

[64] C. Haack and A. Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *LNCS*, pages 202–216. Springer, 2005.

[65] K. Hammond. The dynamic properties of hume: A functionally-based concurrent language with bounded time and space behaviour. In *Proceedings of the 12th International Workshop on the Implementation of Functional Languages (IFL'00)*, volume 2011 of *LNCS*, pages 122–139. Springer, 2000.

[66] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[67] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 48–60. ACM, 2005.

[68] J. Harrison. HOL Light tutorial. Available at http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf, 2014.

[69] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition, 2011.

[70] M. Hennessy. *A Distributed Pi-calculus*. Cambridge University Press, 2007.

[71] M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: A language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

[72] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

[73] I. Herman, B. Adida, M. Sporny, and M. Birbeck. RDFa 1.1 primer - third edition. Available at http://www.w3.org/TR/rdfa-primer/, 2015.

[74] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 235–245, 1973.

[75] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[76] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM, 2003.

[77] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *Proceedings of the 15th European Symposium on Programming (ESOP'06)*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[78] K. Honda. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[79] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[80] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Transactions on Programming Languages and Systems*, 29(6), 2007.

[81] R. Horne and V. Sassone. A verified algebra for read-write Linked Data. *Science of Computer Programming*, 89:2–22, 2014.

[82] R. Horne, V. Sassone, and N. Gibbins. Operational semantics for SPARQL update. In *Proceedings of The Semantic Web - Joint International Semantic Technology Conference, (JIST'11)*, volume 7185 of *LNCS*, pages 242–257. Springer, 2011.

[83] G. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41:37–49, 2007.

[84] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[85] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[86] F. Iwama, A. Igarashi, and N. Kobayashi. Resource usage analysis for a functional language with exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'06)*, pages 38–47. ACM, 2006.

[87] S. Jakšić. Input/output types for dynamic web data, extended abstract. In *Proceedings of the 13th Italian Conference on Theoretical Computer Science (ICTCS'12)*, 2012.

[88] S. Jakšić and L. Padovani. Exception handling for copyless messaging. In *Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'12)*, pages 151–162. ACM, 2012.

[89] S. Jakšić and L. Padovani. Exception handling for copyless messaging. *Science of Computer Programming*, 84:22–51, 2014.

[90] S. Jakšić, J. Pantović, and S. Ghilezan. Linked data privacy. *Mathematical Structures in Computer Science*, FirstView:1–21, 2015.

[91] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP'86)*, pages 105–112. ACM, 1986.

[92] N. Kobayashi. Type systems for concurrent programs. In *Proceedings of the 10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 439–453. Springer, 2002.

[93] N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR'06)*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[94] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the π-calculus. In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 358–371. ACM, 1996.

[95] M. Kolundzija. Security types for sessions and pipelines. In *Revised Selected Papers of the 5th International Workshop Web Services and Formal Methods, (WS-FM'08)*, volume 5387 of *LNCS*, pages 175–190. Springer, 2008.

[96] D. R. Kuhn, E. J. Coyne, and T. R. Weil. Adding attributes to role-based access control. *IEEE Computer*, 43(6):79–81, 2010.

[97] J. Lambek. Pregroup grammars and Chomsky's earliest examples. *Journal of Logic, Language and Information*, 17(2):141–160, 2008.

[98] F. Levi and D. Sangiorgi. Controlling interference in ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[99] H.-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reduce. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 311–318. IEEE, 2002.

[100] T. Maciążek. Content-based information flow verification for C. Master's thesis, DTU, 2015.

[101] F. Manola and E. Miller. RDF primer. Available at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, 2004.

[102] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[103] M. Merro and M. Hennessy. A bisimulation-based semantic theory of safe ambients. *ACM Transactions on Programming Languages and Systems*, 28(2):290–330, 2006.

[104] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 856–867. Springer, 1998.

[105] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[106] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer, 1993.

[107] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[108] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100(1):1–40,41–77, 1992.

[109] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[110] A. Naumowicz and A. Kornilowicz. A brief overview of mizar. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 67–72. Springer, 2009.

[111] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2015.

[112] U. Norell and J. Chapman. Dependently typed programming in agda. Available at http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf, 2012.

[113] S. Osborn, R. S. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[114] J. Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.

[115] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, pages 176–185. IEEE, 1998.

[116] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universitat Hamburg, 1962.

[117] S. L. Peyton Jones, C. D. Clack, J. Salkild, and M. Hardie. GRIP - A High-Performance Architecture for Parallel Graph Reduction. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA'87)*, volume 274 of *LNCS*, pages 98–112. Springer, 1987.

[118] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 202–206. Springer, 1999.

[119] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[120] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[121] C. Priami. Stochastic pi-calculus. *Computer Journal*, 38(7):578–589, 1995.

[122] E. Prud'hommeaux, S. Harris, and A. Seaborne. SPARQL 1.1 query language. Available at http://www.w3.org/TR/2013/REC-sparql11-query-20130321/, 2013.

[123] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

[124] W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[125] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery Update Facility 1.0. Available at https://www.w3.org/TR/xquery-update-10/, 2011.

[126] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. Available at https://www.w3.org/TR/xquery-30/, 2014.

[127] O. Sacco and A. Passant. A privacy preference manager for the social semantic web. In *Proceedings of the 2nd Workshop on Semantic Personalized Information Management: Retrieval and Recommendation*, volume 781 of *CEUR Workshop Proceedings*, pages 42–53. CEUR-WS.org, 2011.

[128] O. Sacco and A. Passant. A privacy preference ontology (PPO) for linked data. In *Proceedings of Workshop on Linked Data on the Web*, volume 813 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[129] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[130] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[131] G. Schreiber and Y. Raimond. RDF 1.1 primer. Available at http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/, 2014. (work in progress).

[132] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[133] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindstrom. JSON-LD 1.0. Available at http://www.w3.org/TR/json-ld/, 2014.

[134] M. Stankovic, A. Passant, and P. Laublet. Directing status messages to their audience in online communities. In *Proceedings of Coordination, Organizations, Institutions and Norms in Agent Systems*, volume 6069 of *LNCS*, pages 195–210. Springer, 2009.

[135] M. Steedman. *The Syntactic Process*. MIT Press, 2000.

[136] Z. Stengel and T. Bultan. Analyzing Singularity channel contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 13–24. ACM, 2009.

[137] K. Suenaga, R. Fukuda, and A. Igarashi. Type-based safe resource deallocation for shared-memory concurrency. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 1–20. ACM, 2012.

[138] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, and J.-K. Zinzindohoue. Dependent types and multi-monadic effects in f. Available at https://www.fstar-lang.org/papers/mumon/paper.pdf, 2015.

[139] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2nd edition, 2006.

[140] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. http://homotopytypetheory.org/book, 2013.

[141] R. H. Thomason, editor. *Formal philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[142] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[143] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 79–88. ACM, 1996.

[144] V. T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[145] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings of the 4th International Conference on Concurrency Theory, (CONCUR'93)*, volume 715 of *LNCS*, pages 524–538. Springer, 1993.

[146] J. Villard, E. Lozes, and C. Calcagno. Proving copyless message passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09)*, volume 5904 of *LNCS*, pages 194–209. Springer, 2009.

[147] J. Villard, E. Lozes, and C. Calcagno. Tracking heaps that hop with Heap-Hop. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, LNCS, pages 275–279. Springer, 2010.

[148] W. Weimer. Exception-handling bugs in java and a language extension to avoid them. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *LNCS*, pages 22–41. Springer, 2006.

[149] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 419–431. ACM, 2004.

[150] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2):1–51, 2008.

[151] A. Westin. *Privacy and Freedom.* Atheneum New York, 1967.

[152] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, 1910–1913.

[153] F. Yang, C. Hankin, F. Nielson, and H. R. Nielson. Predictive access control for distributed computation. *Science of Computer Programming*, 78(9):1264 – 1277, 2013.