

BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

XML-FLUENT MOBILE AGENTS

(Draft)

by

SANTIAGO M. PERICAS-GEERTSEN

B.A., Instituto Tecnológico de Buenos Aires, 1994

M.A., Boston University, 1999

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2001

Approved by

First Reader

Assaf J. Kfoury, Ph.D.
Professor of Computer Science
Boston University

Second Reader

xxxx, Ph.D.
Professor of Computer Science

Third Reader

yyyy, Ph.D.
Professor of Computer Science

XML-FLUENT MOBILE AGENTS

(Order No. nnn)

SANTIAGO M. PERICAS-GEERTSEN

Boston University Graduate School of Arts and Sciences, 2001

Major Professor: Assaf J. Kfoury, Professor of Computer Science, Boston University

ABSTRACT

The Internet is considered to be among the greatest inventions of the twentieth century. Despite its immense importance we, programming language designers, have yet to propose a programming language that can exploit a network that spans the entire globe.

In this thesis, we introduce a language for a programming model based on XML-speaking mobile agents. The thesis is divided into three parts: the first part defines the *mobility* aspect of the language, the second part introduces a typed calculus for manipulating XML documents, and the third part proposes a way of integrating these two calculi. The final goal is to design a typed language in which computation can be carried out by mobile agents exchanging data in the form of XML documents.

The Ambient Calculus was developed by Cardelli and Gordon as a formal framework to study issues of mobility and migrant code [CG98]. In Part I, we consider an extension to the Ambient Calculus where ambients are equipped with a set of local channels over which processes can communicate. This extension incorporates certain features from the π -Calculus, such as the ability to communicate over higher-order channels, but without violating design principles of the Ambient Calculus (e.g., that two processes are unable to communicate if located in different administrative domains). For the extended calculus, we introduce an appropriate operational semantics and a provably sound type system. We conclude this part by showing an encoding of

the Ambient Calculus into our calculus that is both semantics preserving and type preserving.

XML is a simple and portable way of representing structured data in the form of trees. It is an ideal language for machine-to-machine (or peer-to-peer) exchange of data because (i) it is portable, so there is no need to worry about things like integer representations (big endian vs. little endian) or string representations (null terminated vs. fixed length) and (ii) structured data is easily represented in the form of trees. Interestingly, the ability of XML to represent almost any form of structured data is often an important weakness at the same time: for a data exchange to be successful, a producer and a consumer must agree on the structure of the exchanged documents so that the latter can find/extract what it needs. This problem can be addressed by the use of a *type system*. A type system is needed in order to keep track of the meta-data (i.e. data about the data) which can be used by a consumer as a guideline for data extraction. A number of type systems for XML have been proposed in recent years. Perhaps surprisingly, specially if we look at the evolution of programming languages, the use of a type system in XML (as well as in languages for manipulating XML documents) is not mandatory. Domain-specific languages for XML proposed in the last few years are all, with the exception of XDuce [HP00, HP01], *dynamically typed*. Experience has showed us that statically-typed languages are better suited for real-world applications than dynamically-typed ones. Despite requiring more from the programmer's perspective (at least in the absence of a type-inference engine) the benefits greatly outcast the requirements. Specifically, the use of a statically-typed language ensures that the final product is free of an important number of errors (so-called type errors). In Part II, we present the $\delta\lambda$ -Calculus, a statically-typed calculus for manipulating XML documents, and show how it can be integrated with existing XML technology by means of two algorithms for importing and exporting documents into the calculus.

Contents

I	Mobile Agents	1
1	Introduction to Part I	3
1.1	π -Calculus	5
1.2	Ambient Calculus	7
1.3	Channeled Ambient Calculus	10
2	Channeled Ambients	13
2.1	Channeled Ambient Calculus	13
3	Type System	19
3.1	Types and Typing Rules	19
3.2	Subject Reduction	23
4	Encoding the Ambient Calculus	35
4.1	Encoding	35
5	Conclusions to Part I	43
5.1	Discussion and Future Work	43

II XML	47
6 Introduction to Part II	49
6.1 XML	49
6.2 Motivating Example	52
7 Untyped $\delta\lambda$-Calculus	55
7.1 Untyped δ -Calculus	55
7.2 Untyped $\delta\lambda$ -Calculus	59
8 Typed $\delta\lambda$-Calculus	65
8.1 Typed $\delta\lambda$ -Calculus	65
8.2 Type System	66
8.3 Type System with Union Types	69
8.4 Subject Reduction	73
9 From XML to the $\delta\lambda$-Calculus	79
9.1 Documents and DTD Types	80
9.2 Well-formed DTD Types	82
9.3 Algorithm IMP	87
9.4 Correctness of Algorithm IMP	89
9.5 From the $\delta\lambda$ -Calculus to XML	96
10 Recursive Types	99
10.1 Recursive Types	100

11 Conclusions to Part II	109
11.1 Discussion and Future Work	109
12 The DL Language	113
12.1 Syntax	113
12.2 Strings and Labeled Expressions	115
12.3 Tuples	116
12.4 Lists	117
12.5 Variables	118
12.6 Path Expressions	119
12.7 Functions	120
12.8 Types	121
12.9 Injections and Cases	123
12.10 Miscellaneous	123
12.11 An Example	124
12.12 Complete Syntax	127
III XML-Fluent Mobile Agents	135
13 Introduction to Part III	137
14 XML-Fluent Agents	139
14.1 Type System	140
15 Conclusions to Part III	143

List of Figures

2.1	Structural Congruence	15
2.2	Operational Semantics.	16
2.3	Free and Bound Names.	17
3.1	Typing Rules	21
7.1	Operational Semantics: δ -Calculus.	58
7.2	Operational Semantics: CBV λ -Calculus.	60
7.3	Operational Semantics.	61
8.1	Typing Rules for the $\delta\lambda$ -Calculus.	67
8.2	Operational Semantics.	70
8.3	Typing Rules.	71
9.1	Algorithm IMP.	88
9.2	Algorithm IMP'.	92
9.3	Algorithm EXP.	97
10.1	Axiomatization of \approx	102

12.1 Expressions and Values	131
12.2 Operational Semantics	132
12.3 Typing Rules (1 of 3)	133
12.4 Typing Rules (2 of 3)	134
12.5 Typing Rules (3 of 3)	134
14.1 Additional Rules for $\delta\lambda$ -CAC	140

Part I

Mobile Agents

Chapter 1

Introduction to Part I

With the advent of the Internet a few years ago, considerable effort has gone into the study of *mobile computation* and programming languages that support it. On the theoretical side of this research, several concurrent and distributed calculi have been proposed, such as the Distributed Join Calculus [FGL⁺96], the $D\pi$ Calculus [RH98, RH99], the Box-Pi Calculus [SV99], the Seal Calculus [VC99], among others.¹

The Ambient Calculus [Car99] is a recent addition to this list and the starting point of our investigation. Our main interest is the design of a strongly-typed programming language for mobile computation. Part of this effort is an examination of the Ambient Calculus as a foundation for such a language.

The π -Calculus [Mil99] has been used as a model to describe distributed computations. In particular, certain extensions of the π -Calculus [GH99] have been shown adequate to describe network protocols, e.g. client-server interactions. These protocols have been designed under

¹The proliferation of calculi is mostly the result of different concerns and emphases (mobility, concurrency, security, etc.) brought by different researchers. At this early stage of Internet programming, it is perhaps healthy to have several research agendas based on almost as many different calculi.

the assumption that, in principle, channels of communication can be established between any pair of participating peers. However, as explained in [Car99], this assumption fails to hold for wide-area networks, such as the Internet, due to the existence of virtual walls (a.k.a. firewalls) that are erected to protect administrative domains. The π -Calculus relies on the aforementioned assumption because it does not define a notion of *location*, or as we like to call it, the “where”. It is possible to define the location of a process as the set of channels (also known as links) it has available at a particular point in time [Mil99], but this indirect definition only reflects the lack of a first-class notion of location in the π -Calculus.

The Ambient Calculus, on the other hand, has been designed with a different set of *observables* in mind, which imply the possibility that two peers are unable to communicate if located in different administrative domains. To satisfy this new set of observables, it defines a notion of *location* as a first-class entity, so the whereabouts of a process is clearly defined at every point in time. Process communication, on the other hand, is more impersonal: a process enters an ambient and broadcasts a message which is picked up by one (and only one) collocated peer. As a result, the notion of the “whom”, i.e. the receiver of a message, is not as explicit as in the π -Calculus where a process can direct a message to a peer by using a specific channel. This design decision has many implications, among them, the difficulty of creating a type system in which ambients are allowed to be polymorphic [AKPG01, AKPG00].

A wide-area network (WAN) can be seen as a network for interconnecting local-area networks (LANs). As a result, we believe that an appropriate foundational language for Internet programming will combine design principles from both the π -Calculus, which is suited for LAN programming, and the Ambient Calculus, which is suited for WAN programming.

Communication in Ambient Calculus is restricted to processes that belong to the same am-

bient. Consequently, remote communication is achievable only via process migration. It is precisely this feature that makes its integration with the π -Calculus non-trivial. By simply merging the syntax of the Ambient Calculus with that of the π -Calculus, it would be difficult to define the semantics of process communication without violating a design principle of the former.

In the first part of this thesis we present a calculus which embodies what are, in our view, essential features of a core language for programming the Internet: it incorporates the notion of “where” a process is located as well as the notion of with “whom” a process is communicating.² We refer to our calculus as the Channeled Ambient Calculus or **CAC**.

We shall present an example motivating our calculus in Section 1.3. Sections 1.1 and 1.2 are brief introductions to the π -Calculus and the Ambient Calculus, respectively.

1.1 π -Calculus

The π -Calculus is a process calculus where computation is carried out by exchanging messages over named channels [Mil99]. In its most basic formulation, the only messages that can be exchanged are channel names, but it is a simple task to extend the calculus so that other values like integers or booleans can also be communicated.

Two different versions of the π -Calculus can be devised depending on whether an output operation is synchronous or asynchronous. The synchronization of an output operation is not defined with respect to an input process with which it exchanges a message, but with respect to its continuation. That is, an output operation is asynchronous if its continuation is not blocked waiting for the completion of the output. A simple way to define an asynchronous version of

²As explained above, the “whom” refers to the ability of a process to select a channel of communication. Since communication is non-deterministic, the actual receiver of a communication cannot be determined statically.

the π -Calculus is by ensuring that outputs have only empty continuations (i.e. no continuations). This restriction forces a programmer to put continuations in parallel with outputs and, therefore, naturally achieves the asynchronous effect.

The syntax of the asynchronous π -Calculus is shown below. As in other process calculi, there are ways of composing processes $P \mid Q$, replicating processes $!P$ and restricting the scope of a name $(\nu n).P$. The process $M\langle N \rangle$ outputs N over channel M , and the process $M(x).P$ reads a message from channel M and continues as P . Since output processes have no continuations, this formulation of the π -Calculus is, as explained earlier, naturally asynchronous.

P, Q, R	$::=$	$\mathbf{0}$	(Inactivity)
		$ \ !P$	(Replication)
		$ \ P \mid Q$	(Composition)
		$ \ (\nu n).P$	(Restriction)
		$ \ M\langle N \rangle$	(Output)
		$ \ M(x).P$	(Input)
M, N	$::=$	n	(Name)
		$ \ x$	(Variable)

The operational semantics of the π -Calculus is defined based on a congruence relation over processes and the axiom $n\langle n' \rangle \mid n(x).P \longrightarrow P\{x := n'\}$. The reader is referred to [Mil99] for the complete axiomatization.

EXAMPLE 1.1.1. The π -Calculus can be used as a model to describe the interactions of processes running on the same host computer. For the sake of the example, we extend the syntax of expressions in the π -Calculus (ranged by M, N) with a few additional primitives such as strings and if-then-else.

Suppose there is a process that knows the channels over which other processes are accepting requests. Let us refer to this process as the *daemon*. A client process interacts with the daemon

by requesting a specific service by name. The daemon replies by sending a channel over which a process offering that service is expecting connections.

$$\begin{aligned} \text{Server} &= (\text{daemon_rqst}(x).\text{daemon_rply}\langle \text{if } x = \text{'ftp'} \text{ then } \text{ftpd} \text{ else } \dots \rangle \mid \\ &\quad \text{ftpd}(x).P \mid \\ &\quad \text{httpd}(x).Q \mid \dots) \\ \text{Client} &= (\text{daemon_rqst}\langle \text{'ftp'} \rangle \mid \text{daemon_rply}(\text{ftpd}).R) \end{aligned}$$

The process under consideration is $(\text{Server} \mid \text{Client})$. The daemon is modeled by the first line in the definition of *Server*; the other lines in that definition model the processes offering services to clients.

In this example, the client sends the string ‘ftp’ over *daemon_rqst* to the daemon and obtains the channel *ftpd* over *daemon_rply* in response. The assumption is that, after the exchange between the client and the daemon is completed, process *R* interacts directly with process *ftpd(x).P*. □

1.2 Ambient Calculus

The Ambient Calculus was introduced by Cardelli and Gordon in [CG98, Car99]. Despite being a process calculus like the π -Calculus, the model of computation proposed in the Ambient Calculus is quite different: computation is carried out by processes that move about a dynamically changing topology of locations.

An ambient is defined as a place where computation happens. Since ambients are named, they also serve as *locations* for processes residing within their confines. Thus, one key difference between the π -Calculus and the Ambient Calculus is that in the latter the whereabouts of a process, though dynamically changing, is always well defined.

The syntax of the Ambient Calculus is shown below. The first four productions that define the syntax of processes are identical to those in the π -Calculus.

$P, Q, R ::= \mathbf{0}$		(Inactivity)
$!P$		(Replication)
$P \mid Q$		(Composition)
$(\nu n).P$		(Restriction)
$n[P]$		(Ambient)
$M.P$		(Action)
$M, N ::= \text{in } n$		(Enter Ambient)
$\text{out } n$		(Exit Ambient)
$\text{open } n$		(Open Ambient)

An ambient is written as $n[P]$ where n is a name and P is a process. There are three possible actions a process can execute, namely, it can enter an ambient, exit an ambient or open an ambient. These actions are carried out by exercising the capabilities in n , $\text{out } n$ or $\text{open } n$, respectively, as illustrated by the following reduction rules:

$m[\text{in } n.P \mid Q] \mid n[R] \longrightarrow n[m[P \mid Q] \mid R]$	(Red In)
$m[n[\text{out } m.P \mid Q] \mid R] \longrightarrow m[R] \mid n[P \mid Q]$	(Red Out)
$m[\text{open } n.P \mid n[R] \mid Q] \longrightarrow m[P \mid R \mid Q]$	(Red Open)

Notice that in both (Red In) and (Red Out) the ambient that encloses the process exercising the capability is also moved. For this reason, moves in the Ambient Calculus are said to be *subjective*. Other kinds of moves are also plausible; the reader is referred to [CG98] for a discussion of *objective* moves as well as for a complete axiomatization of the reduction relation outlined above.

Although not absolutely necessary from a theoretical point of view—the formulation of the calculus presented above is shown to be Turing complete in [CG98]—communication primitives are often considered as part of the basic Ambient Calculus. Communication is, however, restricted to processes that are collocated, so if two processes that reside on different ambients

wish to communicate, they must first move to a common location before the communication can take place. There are two values that can be communicated in the Ambient Calculus: ambient names and (paths of) capabilities. As in the π -Calculus, communication in the Ambient Calculus is *asynchronous* and, thus, can be viewed (cf. the metaphor in [Car99]) as the placement and subsequent removal of a Post-It note on a message board.

The syntax of the Ambient Calculus is extended to accommodate the communication primitives in the following way:

1. Two new processes are defined, namely, an output process $\langle M \rangle$ and an input process $(x).P$.

Notice that these processes do not specify a channel over which values are read/written.

2. The name n is replaced by the metavariable N in all productions with the exception of $(\nu n).P$, and the syntax of expressions is extended with variables (ranged by x), names (ranged by n) and paths of capabilities (via the constructors ϵ and $M.N$).

The operational semantics is then extended with the axiom $\langle M \rangle \mid (x).P \longrightarrow P\{x := M\}$. Once again, the reader is referred to [CG98] for more details.

There is an important difference between names and capabilities in the Ambient Calculus. A capability is something that, once exercised, is no longer possessed; in particular, there is no way of extracting a name from a capability. Possessing a name, on the other hand, gives us the ability to create any number of ambients and any number of capabilities, so they should be treated more delicately to avoid undesirable consequences (e.g. the unwanted dissolution of an ambient $n[P]$ by an attacker which was able to construct a capability open n after obtaining the name n).

EXAMPLE 1.2.1. Let us now show how to re-write Example 1.1.1 using primitives from the

Ambient Calculus. The server is model by an ambient that includes subambients for the daemon and for each of the processes offering services to clients. As in that example, the process under consideration is $(\text{Server} \mid \text{Client})$.

$$\begin{aligned} \text{Server} &= \text{server}[\text{daemon}[\text{open } \text{client}.0 \mid \\ &\quad (x).(\text{sync}[0] \mid \langle \text{if } x = \text{'ftp'} \text{ then } (\text{in } \text{ftpd}) \text{ else } \dots \rangle)] \mid \\ &\quad \text{ftpd}[\text{open } \text{client}.0 \mid (x).P] \mid \\ &\quad \text{httpd}[\text{open } \text{client}.0 \mid (x).Q] \mid \dots] \\ \text{Client} &= \text{client}[\text{in } \text{server}.\text{in } \text{daemon}.\langle \text{'ftp'} \rangle \mid \\ &\quad \text{open } \text{sync}.(x).\text{client}[\text{out } \text{daemon}.x.R]] \end{aligned}$$

The client moves into the daemon and, after requesting an ftp service, it gets a capability to enter the *ftpd* ambient. Notice the use of an auxiliary ambient named *sync* to synchronize the exchange between the client and the daemon. This synchronization is necessary due to the absence of named channels in the Ambient Calculus (in Example 1.1.1 this problem was avoided by the use of two channels: *daemon_rqst* and *daemon_rply*). Once the client has received the capability, it moves out of the daemon and into the *ftpd* ambient where, after being opened, leaves *R* ready to interact with $(x).P$. □

1.3 Channeled Ambient Calculus

In Examples 1.1.1 and 1.2.1 we showed how to write the same program using, respectively, primitives from the π -Calculus and primitives from the Ambient Calculus.

In Example 1.1.1, the interactions between the client, the daemon and the service provider were easy to model by the use of named channels. The assumption in this case was that it is possible for the client to establish a connection to the server and vice versa, i.e. that no virtual barriers exist between them.

In Example 1.2.1, virtual barriers were not a problem as the client moves into the server, thus translating remote requests into local requests. On the other hand, the interactions between the client, the daemon and the service provider were, due to some limitations in the Ambient Calculus, not as easy to model as in the π -Calculus. In particular, we had to define a few additional ambients to avoid a miscommunication between the processes involved.

Let us show how the same example can be modeled in our calculus. **CAC** extends the Ambient Calculus in the following ways: (i) every ambient declares a set of local channels over which processes can communicate, (ii) inputs and outputs take place over these local channels and (iii) output processes have continuations.

EXAMPLE 1.3.1. The ambient representing the Server declares a number of local channels, among them, *daemon_rqst*, *daemon_rply*, *ftpd* and *httpd*. The Client, which we assume is unaware of all the services provided by the Server, declares only the channels *daemon_rqst* (to send a request to the Server's daemon) and *daemon_rply* (to receive a response from the Server's daemon). Notice that, as in the π -Calculus, it is possible to declare higher-order channels (i.e. channels over which we can send channels) in **CAC**: an example of such a channel is *daemon_rply*.

$$\begin{aligned} \text{Server} = & \text{server}[\text{daemon_rqst}, \text{daemon_rply}, \text{ftpd}, \text{httpd}, \dots ; \\ & \text{daemon_rqst}(x). \text{daemon_rply} \langle \text{if } x = \text{'ftpd'} \text{ then } \text{ftpd} \text{ else } \dots \rangle . \mathbf{0} \quad | \\ & \text{ftpd}(x).P \quad | \\ & \text{httpd}(x).Q \quad | \\ & \dots \quad | \\ & \text{open } \text{client} . \mathbf{0}] \\ \\ \text{Client} = & \text{client}[\text{daemon_rqst}, \text{daemon_rply}; \\ & \text{in } \text{server} . \mathbf{0} \quad | \\ & \text{daemon_rqst} \langle \text{'ftpd'} \rangle . \text{daemon_rply}(x).R] \end{aligned}$$

As explained in Chapter 3, the interaction between Server and Client is *safe* because the latter declares a subset of the channels declared by the former, and because these common channels

12

have identical types.

□

Chapter 2

Channeled Ambients

In this chapter we introduce the syntax and the operational semantics of **CAC**. We consider a version of **CAC** in which input and output operations are unary; we believe, however, that all our results can be extended to the n -ary case without major complications. The operational semantics is defined modulo a congruence relation which is, essentially, an extension of that defined for the Ambient Calculus in [Car99].

2.1 Channeled Ambient Calculus

The syntax of **CAC** is shown below. As in the Ambient Calculus, there are constructs for parallel composition $(P \mid Q)$, name restriction $((\nu n : \sigma).P)$, process replication $(!P)$ and capability execution $(M.P)$.

The main differences between **CAC** and the Ambient Calculus are that in the former (i) every ambient declares a set of local channels over which processes can communicate, (ii) inputs and outputs take place over these local channels and (iii) output is synchronous and have continua-

tions.

$P, Q, R \in \text{Proc}$	$::=$	$\mathbf{0}$	(Inactivity)
		$ P \mid Q$	(Parallel Composition)
		$!P$	(Replication)
		$ (\nu n : \sigma).P$	(Name Restriction)
		$ M.P$	(Action)
		$ M[c_i^{i \in I}; P]$	(Channeled Ambient)
		$ M\langle N \rangle.P$	(Output)
		$ M(n : \sigma).P$	(Input)
$M, N \in \text{PExp}$	$::=$	n	(Name)
		$ c$	(Channel Name)
		$ \text{in } M$	(Enter Capability)
		$ \text{out } M$	(Exit Capability)
		$ \text{open } M$	(Open Capability)
		$ \epsilon$	(Null Path)
		$ M.N$	(Composite Path)

The main differences between **CAC** and the Ambient Calculus are that in the former (i) every ambient declares a set of local channels over which processes can communicate, (ii) inputs and outputs take place over these local channels and (iii) output is synchronous and have continuations.

The set of expressions PExp, ranged by M and N , includes ambient names n , channel names c and (path of) capabilities. There are two binding constructs for names: $(\nu n : \sigma).P$ and $M(n : \sigma).P$. In the latter, a name plays the role of a variable as we shall see shortly when we present the operational semantics. For this reason, we sometimes use x, y and z to range over input bound names as well. In what follows, we assume that the set of ambient names and the set of channel names are disjoint.

An ambient in **CAC** is written as $M[c_i^{i \in I}; P]$, where M is an expression (typically a name), c_i with $i \in I$ is a set of channel names and P is a process. Note that all name binding constructs

$P \mid Q \equiv Q \mid P$	(Struct ParComm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct ParAssoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n : \tau).(\nu m : \sigma).P \equiv (\nu m : \sigma).(\nu n : \tau).P \quad \text{if } n \neq m$	(Struct ResRes)
$(\nu n : \tau).(P \mid Q) \equiv P \mid (\nu n : \tau).Q \quad \text{if } n \neq \text{fn}(P)$	(Struct ResPar)
$(\nu n : \tau).m[c_i^{i \in I}; P] \equiv m[c_i^{i \in I}; (\nu n : \tau).P] \quad \text{if } n \neq m$	(Struct ResAmb)
$P \mid \mathbf{0} \equiv P$	(Struct ZeroPar)
$(\nu n : \tau).\mathbf{0} \equiv \mathbf{0}$	(Struct ZeroRes)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct ZeroRepl)
$\epsilon.P \equiv P$	(Struct ϵ)
$(M.N).P \equiv M.N.P$	(Struct .)
$P \equiv Q \Rightarrow (\nu n : \tau).P \equiv (\nu n : \tau).Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow M[c_i^{i \in I}; P] \equiv M[c_i^{i \in I}; Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \equiv Q \Rightarrow M(n : \tau).P \equiv M(n : \tau).Q$	(Struct Input)
$P \equiv Q \Rightarrow M\langle N \rangle.P \equiv M\langle N \rangle.Q$	(Struct Output)
$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)

Figure 2.1. Structural Congruence

in **CAC** have type annotations; the syntax of types is introduced in Chapter 3. When there is no ambiguity, we write M as a shorthand for the process $M.\mathbf{0}$.

Like in the Ambient Calculus, the operational semantics of **CAC** is defined modulo a congruence relation \equiv . This congruence relates processes that, although syntactically different, denote the same logical computation. In addition to the rules shown in Figure 2.1, we identify terms modulo α -renaming and reordering of channel declarations inside ambients. For example, assuming m and y not free in P , we have

$m[c_i^{i \in I}; \text{in } n.P \mid Q] \mid n[c_j^{j \in J}; R] \longrightarrow n[c_j^{j \in J}; m[c_i^{i \in I}; P \mid Q] \mid R]$	(Red In)
$m[c_i^{i \in I}; n[c_j^{j \in J}; \text{out } m.P \mid Q] \mid R] \longrightarrow m[c_i^{i \in I}; R] \mid n[c_j^{j \in J}; P \mid Q]$	(Red Out)
$m[c_i^{i \in I}; \text{open } n.P \mid n[c_j^{j \in J}; R] \mid Q] \longrightarrow m[c_i^{i \in I}; P \mid R \mid Q]$	(Red Open)
$m[c_i^{i \in I}; c_k \langle M \rangle . R \mid c_k(n : \sigma).P \mid Q] \longrightarrow m[c_i^{i \in I}; R \mid P\{n := M\} \mid Q] \quad \text{if } k \in I$	(Red Comm)
$P \longrightarrow Q \Rightarrow n[c_i^{i \in I}; P] \longrightarrow n[c_i^{i \in I}; Q]$	(Red Amb)
$P \longrightarrow Q \Rightarrow (\nu n : \sigma).P \longrightarrow (\nu n : \sigma).Q$	(Red Res)
$P \longrightarrow Q \Rightarrow P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \Rightarrow P' \longrightarrow Q'$	(Red \equiv)

Figure 2.2. Operational Semantics.

$$(\nu n : \tau).n[c_2, c_1, c_3; c_1(x : \sigma).P] \equiv (\nu m : \tau).m[c_1, c_2, c_3; c_1(y : \sigma).P\{n := m\}\{x := y\}].$$

The semantics of **CAC** is presented in Figure 2.2. We write \longrightarrow for the reflexive and transitive closure of \longrightarrow , and \longrightarrow^{\equiv} for the union of the binary relations \longrightarrow and \equiv .

Notice that in addition to (Red In) and (Red Out) as in [CG99], both (Red Open) and (Red Comm) demand the existence of an enclosing ambient for the reduction step to take place. Moreover, the rule (Red Comm) requires the presence of a local declaration for the channel over which communication happens (this is enforced by the side condition $k \in I$). Consider the process

$$Q = h[c_1; \text{open } p] \mid p[; \text{in } h \mid c_1 \langle M \rangle . \mathbf{0} \mid c_1(x : \sigma). \mathbf{0}].$$

Since ambient p does not declare c_1 as a local channel, the processes $c_1 \langle M \rangle . \mathbf{0}$ and $c_1(x : \sigma). \mathbf{0}$ cannot interact. However, these processes are not stuck forever, as they can still interact if p is opened inside an ambient in which c_1 is declared. Specifically, in this example, the two processes can interact after p is opened inside h —notice that process Q reduces to $h[c_1; \mathbf{0}]$.

The last example shows that a process can refer to a channel that is not declared by its enclosing ambient. We say that a channel is *used* within an ambient if its either declared, i.e. listed

$\text{fn}(\mathbf{0}) = \mathbf{0}$	$\text{bn}(\mathbf{0}) = \mathbf{0}$
$\text{fn}(P \mid Q) = \text{fn}(P) \cup \text{fn}(Q)$	$\text{bn}(P \mid Q) = \text{bn}(P) \cup \text{bn}(Q)$
$\text{fn}(!P) = \text{fn}(P)$	$\text{bn}(!P) = \text{bn}(P)$
$\text{fn}((\nu n : \sigma).P) = \text{fn}(P) - \{n\}$	$\text{bn}((\nu n : \sigma).P) = \text{bn}(P) \cup \{n\}$
$\text{fn}(M.P) = \text{fn}(M) \cup \text{fn}(P)$	$\text{bn}(M.P) = \text{bn}(M) \cup \text{bn}(P)$
$\text{fn}(M[c_i^{i \in I}; P]) = \text{fn}(M) \cup \text{fn}(P)$	$\text{bn}(M[c_i^{i \in I}; P]) = \text{bn}(M) \cup \text{bn}(P)$
$\text{fn}(M(n : \sigma).P) = \text{fn}(M) \cup (\text{fn}(P) - \{n\})$	$\text{bn}(M(n : \sigma).P) = \text{bn}(M) \cup \text{bn}(P) \cup \{n\}$
$\text{fn}(M\langle N \rangle.P) = \text{fn}(M) \cup \text{fn}(N) \cup \text{fn}(P)$	$\text{bn}(M\langle N \rangle.P) = \text{bn}(M) \cup \text{bn}(N) \cup \text{bn}(P)$
$\text{fn}(n) = \{n\}$	$\text{bn}(n) = \emptyset$
$\text{fn}(c) = \emptyset$	$\text{bn}(c) = \emptyset$
$\text{fn}(\text{in } M) = \text{fn}(M)$	$\text{bn}(\text{in } M) = \text{bn}(M)$
$\text{fn}(\text{out } M) = \text{fn}(M)$	$\text{bn}(\text{out } M) = \text{bn}(M)$
$\text{fn}(\text{open } M) = \text{fn}(M)$	$\text{bn}(\text{open } M) = \text{bn}(M)$
$\text{fn}(\epsilon) = \emptyset$	$\text{bn}(\epsilon) = \emptyset$
$\text{fn}(M.N) = \text{fn}(M) \cup \text{fn}(N)$	$\text{bn}(M.N) = \text{bn}(M) \cup \text{bn}(N)$

Figure 2.3. Free and Bound Names.

to the left of “;”, or referred by a top-level process. It follows from this definition that the set of *declared* channels is always a subset of the set of *used* channels.

We denote by $\text{fn}(P)$ the set of names occurring free in P , and by $\text{bn}(P)$ the set of names occurring bound in P . The definitions of $\text{fn}()$ and $\text{bn}()$ are standard (cf. Figure 2.3).

Additionally, we define the set of free channel names in a process P by $\text{fcn}(P)$. The definition of $\text{fcn}()$ is straightforward: all channel names are consider to be free. The rationale for this definition is that if a channel name is not free then it is bound and, as a result, can be consistently renamed. However, this is not the case in **CAC** because renaming channel names may change the meaning of a process. For example, consider the process

$$P = n[c_1; \text{open } m.c_1(x : \sigma).\mathbf{0} \mid m[c_1; c_1\langle M \rangle.\mathbf{0}]].$$

Clearly $P \longrightarrow n[c_1; \mathbf{0}]$, but if we consistently rename c_1 inside m (say to c_2) then there will be no communication after m is opened, so P will no longer reduce to $n[c_1; \mathbf{0}]$.

We can set up an analogy between channels in **CAC** and ports in TCP/IP. The name of a channel in **CAC** (resp. a port number in TCP/IP) is regarded as “common knowledge”. An ambient cannot restrict the scope of a channel, but it has the ability to either activate or deactivate a channel via the use of declarations.

EXAMPLE 2.1.1. Using rules from Figures 2.1 and 2.2, plus a few additional ones to evaluate constants and primitive operations such as if-then-else (cf. Chapter 7), Example 1.3.1 evaluates as follows:

$$\begin{aligned}
 \text{Server} \mid \text{Client} &\longrightarrow^{\equiv} \text{server}[\text{daemon_rqst}, \text{daemon_rply}, \text{ftpd}, \text{httpd}, \dots ; \\
 &\quad \text{daemon_rqst}(x).\text{daemon_rply}\langle \text{if } x = \text{'ftp'} \text{ then } \text{ftpd} \text{ else } \dots \rangle. \mathbf{0} \mid \\
 &\quad \text{ftpd}(x).P \mid \text{httpd}(x).Q \mid \dots \mid \\
 &\quad \text{daemon_rqst}\langle \text{'ftp'} \rangle.\text{daemon_rply}(x).R] \\
 &\longrightarrow^{\equiv} \text{server}[\text{daemon_rqst}, \text{daemon_rply}, \text{ftpd}, \text{httpd}, \dots ; \\
 &\quad \text{ftpd}(x).P \mid \text{httpd}(x).Q \mid \dots \mid \\
 &\quad R\{x := \text{ftpd}\}]
 \end{aligned}$$

Notice that empty continuations $\mathbf{0}$ in parallel with other processes can be discarded using the rule (Struct ZeroPar). □

Chapter 3

Type System

In this chapter we present a type system for our calculus. The type system we propose is based on that introduced in [CG99] for the Ambient Calculus.

The syntax of **CAC** from Chapter 2 allows for so-called meaningless terms like $(\text{in } n)[c_i^{i \in I}; P]$ to be defined. By using our type system, we can guarantee (i) that these syntactic anomalies are not typable and (ii) that typable terms never evaluate to syntactic anomalies.

3.1 Types and Typing Rules

The syntax of types is shown below. Types are divided into process types and expression types, the latter being the types of terms that can be input and output over channels.

$$\begin{array}{lll} \rho \in \text{ProcType} & ::= & \text{pro}[c_i : \wedge \tau_i]^{i \in I} & \text{(Process Type)} \\ \rho, \sigma, \tau, \nu \in \text{PExpType} & ::= & \text{amb}[c_i : \wedge \tau_i]^{i \in I} & \text{(Ambient Type)} \\ & | & \text{cap}[c_i : \wedge \tau_i]^{i \in I} & \text{(Capability Type)} \\ & | & \wedge \sigma & \text{(Channel Type)} \end{array}$$

We write Type for the set $\text{ProcType} \cup \text{PExpType}$, and use the metavariable ρ to range over this

set; the metavariables σ, τ and v range over the set PExpType.

A process type is of the form $\text{pro}[c_i : \wedge \tau_i]^{i \in I}$ where the c_i 's are channel names and the τ_i 's expression types. If a process P is given the type $\text{pro}[c_i : \wedge \tau_i]^{i \in I}$, then c_i with $i \in I$ denotes the set of channels that are used in the ambient in which P resides. An expression type is either the type of an ambient name $\text{amb}[c_i : \wedge \tau_i]^{i \in I}$, or the type of a capability $\text{cap}[c_i : \wedge \tau_i]^{i \in I}$ (resp. a capability path) or the type of a channel $\wedge \sigma$. Notice that the type of each channel in an ambient, capability or process type is prefixed by the type constructor “ \wedge ”.

Let E range over (ambient) name environments and F over channel environments. A *type environment* is a pair $E; F$ of a name environment and a channel environment. Based on our assumption that ambient names and channel names are disjoint, a type environment $E; F$ can be regarded as a partial mapping between names (of either kind) and elements of PExpType. In what follows, we will often refer to a type environment simply as an environment.

The typing rules for **CAC** are shown in Figure 3.1. The type system is inspired by that in [CG99]. Instead of keeping track of a single topic of conversation for each ambient, we must keep track of a topic of conversation for each channel used within an ambient. If an ambient n is opened inside an ambient m , then the set of channels used in n must be a subset of those used in m , and the common channels must agree on their topic of conversation.

The rules (Exp In), (Exp Out) and (Exp Open) are similar to those in [CG99], extended for **CAC**. The first two of these rules do not impose any restrictions on the channel types of the conclusion (this is because moves in **CAC**, as in **AC**, are “subjective”; the reader is referred to [Car99] for a discussion). The rule (Exp Open), on the other hand, requires the channel types in the conclusion to be identical to those in the premise.

In (Proc Amb) the channel types in $\text{amb}[c_i : \wedge \sigma_i]^{i \in I}$, the type of M , are used as the channel

Environments		
(Env \emptyset)	(Env n)	(Env c)
$\frac{}{\emptyset; \emptyset \vdash \diamond}$	$\frac{E; F \vdash \diamond \quad n \notin \text{dom}(E)}{E, n : \tau; F \vdash \diamond}$	$\frac{E; F \vdash \diamond \quad c \notin \text{dom}(F)}{E; F, c : \wedge \tau \vdash \diamond}$
Expressions		
(Exp n)	(Exp c)	(Exp ϵ)
$\frac{E, n : \tau, E'; F \vdash \diamond}{E, n : \tau, E'; F \vdash n : \tau}$	$\frac{E; F, c : \wedge \tau, F' \vdash \diamond}{E; F, c : \wedge \tau, F' \vdash c : \wedge \tau}$	$\frac{E; F \vdash \diamond}{E; F \vdash \epsilon : \text{cap}[c_i : \wedge \tau_i]^{i \in I}}$
(Exp In)	(Exp Out)	
$\frac{E; F \vdash M : \text{amb}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash \text{in } M : \text{cap}[c_j : \wedge \sigma_j]^{j \in J}}$	$\frac{E; F \vdash M : \text{amb}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash \text{out } M : \text{cap}[c_j : \wedge \sigma_j]^{j \in J}}$	
(Exp Open)	(Exp Action)	
$\frac{E; F \vdash M : \text{amb}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash \text{open } M : \text{cap}[c_i : \wedge \tau_i]^{i \in I}}$	$\frac{E; F \vdash M : \text{cap}[c_i : \wedge \tau_i]^{i \in I} \quad E; F \vdash N : \text{cap}[c_j : \wedge \tau_j]^{j \in J}}{E; F \vdash M.N : \text{cap}[c_j : \wedge \tau_j]^{j \in J}} \quad (I \subseteq J)$	
Processes		
(Proc Zero)	(Proc Par)	
$\frac{E; F \vdash \diamond}{E; F \vdash \mathbf{0} : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}$	$\frac{E; F \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I} \quad E; F \vdash Q : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash P \mid Q : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}$	
(Proc Repl)	(Proc Res)	
$\frac{E; F \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash !P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}$	$\frac{E, n : \text{amb}[c_i : \wedge \tau_i]^{i \in I}; F \vdash P : \text{pro}[c_j : \wedge \sigma_j]^{j \in J}}{E; F \vdash (\nu n : \text{amb}[c_i : \wedge \tau_i]^{i \in I}).P : \text{pro}[c_j : \wedge \sigma_j]^{j \in J}}$	
(Proc Amb)		
$\frac{E; F \vdash M : \text{amb}[c_i : \wedge \sigma_i]^{i \in I} \quad E; c_i : \wedge \sigma_i \vdash P : \text{pro}[c_i : \wedge \sigma_i]^{i \in I}}{E; F \vdash M[c_k^{k \in K}; P] : \text{pro}[c_j : \wedge \tau_j]^{j \in J}} \quad (K \subseteq I)$		
(Proc Action)		
$\frac{E; F \vdash M : \text{cap}[c_i : \wedge \tau_i]^{i \in I} \quad E; F \vdash P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}}{E; F \vdash M.P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}} \quad (I \subseteq J)$		
(Proc Input)		
$\frac{E; F \vdash M : \wedge \sigma \quad E, n : \sigma; F \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash M(n : \sigma).P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}$		
(Proc Output)		
$\frac{E; F \vdash M_1 : \wedge \sigma \quad E; F \vdash M_2 : \sigma \quad E; F \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}{E; F \vdash M_1 \langle M_2 \rangle . P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}}$		

Figure 3.1. Typing Rules

environment in which P is typed. This is the reason why channels are said to be “local”: their use is confined to the ambient in which they are declared.¹ In fact, it is possible for two ambients, as long as they have different names, to declare the same channel with different types. Type safety is not compromised by this decision as long as neither ambient is opened inside the other.

Also in (Proc Amb), the side condition $K \subseteq I$ guarantees that the type of every channel declared in an ambient is present in the type of the ambient. Notice that forcing $K = I$ is too restrictive, since we may define ambients in which processes communicate over undeclared channels. In other words, we must keep track of the *used* channels instead of just the *declared* channels.

The rule (Proc Action), by means of its side condition $I \subseteq J$, does not demand the set of channels in $\text{cap}[c_i : \wedge \tau_i]^{i \in I}$ (the type of M) to be identical to that in $\text{pro}[c_j : \wedge \tau_j]^{j \in J}$ (the type of P). This flexibility is needed for a process to open an ambient in which only a subset of the channels present in its enclosing ambient is used. For example, let $\sigma_1 = \text{amb}[c_1 : \wedge \tau_1]$ and $\sigma_2 = \text{amb}[c_1 : \wedge \tau_1, c_2 : \wedge \tau_2]$ in

$$(\nu m : \sigma_1).(\nu n : \sigma_2).n[c_1, c_2; \text{open } m.\mathbf{0} \mid m[c_1; R]]$$

for some types τ_1 and τ_2 . Had the rule (Proc Action) forced $I = J$, this process would have been untypable, as $\text{open } m$ has type $\text{cap}[c_1 : \wedge \tau_1]$ and $\mathbf{0}$ must be assigned the type $\text{pro}[c_1 : \wedge \tau_1, c_2 : \wedge \tau_2]$ in accordance to (Proc Amb). The side condition $I \subseteq J$ in (Exp Action) serves a similar purpose and is also needed for Lemma 3.2.7.²

The types of the channels in (Proc Input) and (Proc Output) appear to be disconnected from

¹This “locality” does not imply that these channels are dummy. That is, the meaning of a process may change if channels are renamed inside an ambient. See Chapter 2 for an example.

²Another way of explaining the nature of these side conditions is to say that the rules (Proc Action) and (Exp Action) “inline” a subtyping relation over $\text{cap}[]$ and $\text{pro}[]$ types. We opted for a syntax-directed version of the type system as this simplifies some of the proofs that are presented later in this chapter.

those in their continuations (denoted by the process P in both rules). However, this is not the case: they are related by the rightmost premise of (Proc Amb), which requires the channels in the process type to be identical to those in the channel environment.

EXAMPLE 3.1.1. Let us now show how to type Example 1.3.1 using the rules presented in this section. Define $Server'$ and $Client'$ to be the typed-annotated versions of $Server$ and $Client$:

$$\begin{aligned}
 Server' &= server[daemon_rqst, daemon_rply, ftpd, httpd, \dots ; \\
 &\quad daemon_rqst(x : string).daemon_rply\langle if\ x = 'ftp' \text{ then } ftpd \text{ else } \dots \rangle.0 \quad | \\
 &\quad ftpd(x : \tau).P \quad | \\
 &\quad httpd(x : \tau).Q \quad | \\
 &\quad \dots \quad | \\
 &\quad open\ client.0] \\
 \\
 Client' &= client[daemon_rqst, daemon_rply; \\
 &\quad in\ server.0 \quad | \\
 &\quad daemon_rqst\langle 'ftp' \rangle.daemon_rply(x : \wedge\wedge\tau).R]
 \end{aligned}$$

The notation “ \dots ” is used to indicate that other services may be offered by $Server'$. It is easy to verify that, given $Server'$ and $Client'$, the judgement shown below is derivable using the rules from Figure 3.1.

$$\begin{aligned}
 E \vdash & (\nu\ server : amb[daemon_rqst : \wedge string, daemon_rply : \wedge\wedge\tau, ftpd : \wedge\tau, httpd : \wedge\tau, \dots]). \\
 & (\nu\ client : amb[daemon_rqst : \wedge string, daemon_rply : \wedge\wedge\tau]).(Server' \mid Client') : pro[]
 \end{aligned}$$

Notice that the fact that channel $daemon_rply$ is a higher-order channel is reflected in the type $\wedge\wedge\tau$, which is the type of channels over which channels of type τ can be exchanged. \square

3.2 Subject Reduction

In this section we prove a Subject Congruence and a Subject Reduction theorem for **CAC**. The lemmas that follow are needed in the proofs of those theorems.

Lemma 3.2.1 (Typing Ambients). *For every channel environment F' and every type $\text{pro}[c_j : \wedge \sigma_j]^{j \in J}$, if $E; F \vdash n[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then $E; F' \vdash n[c_i^{i \in I}; P] : \text{pro}[c_j : \wedge \sigma_j]^{j \in J}$. \square*

Proof. The channel environment F in the judgement $E; F \vdash n[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ is not used to type the term $n[c_i^{i \in I}; P]$. The typing rule (Proc Amb) does not impose any restrictions on the process type chosen for the conclusion. \square

Lemma 3.2.2 (Swapping). *Let U range over the set of expressions and the set of processes. Then,*

1. *If $E, n_1 : \tau_1, n_2 : \tau_2, E'; F \vdash U : \rho$ then $E, n_2 : \tau_2, n_1 : \tau_1, E'; F \vdash U : \rho$,*

2. *If $E; F, c_1 : \wedge \tau_1, c_2 : \wedge \tau_2, F' \vdash U : \rho$ then $E; F, c_2 : \wedge \tau_2, c_1 : \wedge \tau_1, F' \vdash U : \rho$. \square*

Proof. Easy induction on derivations. \square

Lemma 3.2.3 (Weakening). *Let U range over the set of expressions and the set of processes. Then,*

1. *If $E; F \vdash U : \rho$ and $E, n : \tau; F \vdash \diamond$ then $E, n : \tau; F \vdash U : \rho$,*

2. *If $E; F \vdash U : \rho$ and $E; F, c : \wedge \tau \vdash \diamond$ then $E; F, c : \wedge \tau \vdash U : \rho$. \square*

Proof. By induction on derivations using Lemma 3.2.2. \square

Lemma 3.2.4 (Strengthening). *Let P be a process and n be a name such that $n \notin \text{fn}(P)$. If*

$E, n : \tau; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. \square

Proof. By induction on derivations. \square

Lemma 3.2.5. *Let \hat{P} be a process. If $E; c_i : \wedge \tau_i \vdash \hat{P} : \text{pro}[c_i : \wedge \tau_i]^{i \in I}$ and $E; c_j : \wedge \tau_j \vdash \diamond$ with $I \subseteq J$*

then $E; c_j : \wedge \tau_j \vdash \hat{P} : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$. \square

Proof. By induction on derivations. The case for (Proc Zero) is immediate from the hypothesis. The cases for (Proc Par), (Proc Repl) and (Proc Res) follow directly from the induction hypothesis. The rest of the cases are shown next.

(Proc Amb) A consequence of Lemma 3.2.1.

(Proc Action) This judgement must have been derived as follows, with $K \subseteq I$,

$$\begin{array}{rcl}
 E; c_i : \wedge \tau_i \vdash M.P : \text{pro}[c_i : \wedge \tau_i]^{i \in I} & \text{(Proc Action)} & \\
 \uparrow E; c_i : \wedge \tau_i \vdash M : \text{cap}[c_k : \wedge \tau_k]^{k \in K} & \dots & (\dagger) \\
 \uparrow \dots & & \\
 \uparrow E; c_i : \wedge \tau_i \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I} & \dots & (\ddagger) \\
 \uparrow \dots & &
 \end{array}$$

Applying the induction hypothesis to (\ddagger) we have $E; c_j : \wedge \tau_j \vdash P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$. From (\dagger) and Lemma 3.2.3 we derive $E; c_j : \wedge \tau_j \vdash M : \text{cap}[c_k : \wedge \tau_k]^{k \in K}$. By the last two judgements, and $K \subseteq I \subseteq J$, we conclude using (Proc Action) that $E; c_j : \wedge \tau_j \vdash M.P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$ as desired.

(Proc Input) By assumption $E; c_i : \wedge \tau_i \vdash M(n : \sigma).P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}$ and then $E; c_i : \wedge \tau_i \vdash M : \wedge \sigma$ and $E, n : \sigma; c_i : \wedge \tau_i \vdash P : \text{pro}[c_i : \wedge \tau_i]^{i \in I}$. By induction hypothesis $E, n : \sigma; c_j : \wedge \tau_j \vdash P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$, and by Lemma 3.2.3 we have $E; c_j : \wedge \tau_j \vdash M : \wedge \sigma$. Hence, it follows by (Proc Input) that $E; c_j : \wedge \tau_j \vdash M(n : \sigma).P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$ as desired.

(Proc Output) Similar to (Proc Input), using Lemma 3.2.3 and the induction hypothesis. \square

Lemma 3.2.6 (Substitution). *Let \hat{P} be a process and \hat{M} an expression. If $E, x : \sigma, E'; F \vdash \hat{P} : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$ and $E, E'; F \vdash \hat{M} : \sigma$ then $E, E'; F \vdash \hat{P}\{x := \hat{M}\} : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$.* \square

Proof. By induction on derivations. \square

Lemma 3.2.7 (Subject Congruence). *Let \hat{P} and \hat{Q} be processes such that $\hat{P} \equiv \hat{Q}$. Then $E; F \vdash \hat{P} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ if and only if $E; F \vdash \hat{Q} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. \square*

Proof. By induction on derivations based on the rules in Figure 2.1. The case for (Struct Refl) is immediate. The proof for all cases having a non-empty set of premises follows directly from the induction hypotheses.

(Struct ParComm) Immediate from rule (Proc Par).

(Struct ParAssoc) Suppose $E; F \vdash (P \mid Q) \mid R : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. It follows that $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ and $E; F \vdash Q \mid R : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. From the last two judgements using (Proc Par) we have $E; F \vdash Q \mid R : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Thus, from the last judgement and $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ we conclude, once again by (Proc Par), that $E; F \vdash P \mid (Q \mid R) : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired. The same reasoning applies to the converse.

(Struct Repl Par) Suppose $E; F \vdash !P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This implies by (Proc Repl) that $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. From the last two judgement using (Proc Par) we have $E; F \vdash P \mid !P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Conversely, if $E; F \vdash P \mid !P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then by (Proc Par) we have $E; F \vdash !P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Struct ResRes) Suppose $E; F \vdash (\nu n : \tau).(\nu m : \sigma).P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ with $n \neq m$. It follows by two applications of (Proc Res) that $E, n : \tau, m : \sigma; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Using Lemma 3.2.2 and the assumption that $n \neq m$, we have $E, m : \sigma, n : \tau; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Thus, by two applications of (Proc Res) we get $E; F \vdash (\nu m : \sigma).(\nu n : \tau).P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired. The same reasoning applies to the converse.

(Struct ResPar) Suppose $E; F \vdash (\nu n : \tau).(P \mid Q) : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ with $n \notin \text{fn}(P)$. Then it follows that $E, n : \tau; F \vdash P \mid Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ and also that $E, n : \tau; F \vdash Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ and $E, n : \tau; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. From the last judgement, using Lemma 3.2.4, we have $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Thus, using (Proc Res) and (Proc Par) in that order, we conclude that $E; F \vdash P \mid (\nu n : \tau).Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. The converse is similar except for the use of Lemma 3.2.3 instead of Lemma 3.2.4.

(Struct ResAmb) Suppose $E; F \vdash (\nu n : \tau).m[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ with $n \neq m$. This judgement must have been derived as follows:

$$\begin{array}{rcl}
E; F \vdash (\nu n : \tau).m[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} & & \text{(Proc Res)} \\
\uparrow E, n : \tau; F \vdash m[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} & & \text{(Proc Amb)} \\
\uparrow E, n : \tau; F \vdash m : \text{pro}[c_j : \wedge \tau_j]^{j \in J} & & \text{(Exp } n) \quad (\dagger) \\
\uparrow \dots & & \\
\uparrow E, n : \tau; c_j : \wedge \tau_j \vdash P : \text{pro}[c_j : \wedge \tau_j]^{j \in J} & \dots & (\ddagger) \\
\uparrow \dots & &
\end{array}$$

From (\dagger) using Lemma 3.2.4 and the assumption that $n \neq m$ we have $E; F \vdash m : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$. From (\ddagger) applying the rule (Proc Res) we get $E; c_j : \wedge \tau_j \vdash (\nu n : \tau).P : \text{pro}[c_j : \wedge \tau_j]^{j \in J}$. Thus, using the last two judgements we derive $E; F \vdash m[c_i^{i \in I}; (\nu n : \tau).P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. The converse is similar except for the use of Lemma 3.2.3 instead of Lemma 3.2.4.

(Struct ZeroPar) Immediate as $\mathbf{0}$ can be assigned any process type.

(Struct ZeroRes) A consequence of Lemmas 3.2.4 and 3.2.3.

(Struct ZeroRepl) Immediate using rule (Proc Repl).

(Struct ϵ) Suppose $E; F \vdash \epsilon.P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This implies by (Proc Action) that $E; F \vdash$

$P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Conversely, if $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then by (Exp ϵ) we have $E; F \vdash \epsilon : \text{cap}[c_k : \wedge \tau_k]^{k \in K}$, and then by (Proc Action) $E; F \vdash \epsilon.P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Struct .) Suppose $E; F \vdash (M.M').P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This judgement must have been derived as follows:

$$\begin{array}{rcl}
E; F \vdash (M.M').P : \text{pro}[c_k : \wedge \tau_k]^{k \in K} & & \text{(Proc Action)} \\
\uparrow E; F \vdash M.M' : \text{cap}[c_i : \wedge \tau_i]^{i \in I} \quad (I \subseteq K) & & \text{(Exp Action)} \\
\uparrow E; F \vdash M : \text{cap}[c_j : \wedge \tau_j]^{j \in J} \quad (J \subseteq I) & \dots & (\dagger) \\
\uparrow \dots & & \\
\uparrow E; F \vdash M' : \text{cap}[c_i : \wedge \tau_i]^{i \in I} & \dots & (\ddagger) \\
\uparrow \dots & & \\
\uparrow E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K} & \dots & (b) \\
\uparrow \dots & &
\end{array}$$

By (\ddagger) and (b) and $I \subseteq K$, using (Proc Action), we have $E; F \vdash M'.P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$.

By (\dagger) and $J \subseteq I \subseteq K$, using (Proc Action) once again, we conclude $E; F \vdash M.M'.P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired. The same reasoning applies to the converse. \square

Theorem 3.2.8 (Subject Reduction). *Let \hat{P} and \hat{Q} be processes. If $\hat{P} \longrightarrow \hat{Q}$ and $E; F \vdash \hat{P} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then $E; F \vdash \hat{Q} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$.* \square

Proof. By induction on derivations using the rules from Figure 2.2.

(Red In) By assumption we have $m[c_i^{i \in I}; \text{in } n.P \mid Q] \mid n[c_j^{j \in J}; R] \longrightarrow n[c_j^{j \in J}; m[c_i^{i \in I}; P \mid Q] \mid R]$

and $E; F \vdash m[c_i^{i \in I}; \text{in } n.P \mid Q] \mid n[c_j^{j \in J}; R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Then, it must be $E; F \vdash$

$m[c_i^{i \in I}; \text{in } n.P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ and $E; F \vdash n[c_j^{j \in J}; R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. The last

two judgements must have been derived as follows:

$$\begin{array}{l}
E; F \vdash m[c_i^{i \in I}; \text{in } n.P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow \\
E; F \vdash m : \text{amb}[c_l : \wedge v_l]^{l \in L} \quad (I \subseteq L) \quad (\text{Exp } n) \\
\uparrow \\
\vdots \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{in } n.P \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Par}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{in } n.P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Action}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{in } n : \text{cap}[c_g : \wedge v_g]^{g \in G} \quad (G \subseteq L) \quad (\text{Exp In}) \\
\uparrow \\
\vdots \\
\uparrow \\
E; c_l : \wedge v_l \vdash P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (\dagger) \\
\uparrow \\
\vdots \\
\uparrow \\
E; c_l : \wedge v_l \vdash Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (\ddagger) \\
\uparrow \\
\vdots
\end{array}$$

$$\begin{array}{l}
E; F \vdash n[c_j^{j \in J}; R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow \\
E; F \vdash n : \text{amb}[c_h : \wedge \sigma_h]^{h \in H} \quad (J \subseteq H) \quad (\text{Exp } n) \\
\uparrow \\
\vdots \\
\uparrow \\
E; c_h : \wedge \sigma_h \vdash R : \text{pro}[c_h : \wedge \sigma_h]^{h \in H} \quad \dots \quad (b) \\
\uparrow \\
\vdots
\end{array}$$

From (\dagger) and (\ddagger) using (Proc Par) we have $E; c_l : \wedge v_l \vdash P \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L}$, and from $I \subseteq L$ using (Proc Amb) we derive $E; F \vdash m[c_i^{i \in I}; P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. By Lemma 3.2.1, we get $E; c_h : \wedge \sigma_h \vdash m[c_i^{i \in I}; P \mid Q] : \text{pro}[c_h : \wedge \sigma_h]^{h \in H}$. Therefore, from (b) using (Proc Par) we have

$$E; c_h : \wedge \sigma_h \vdash m[c_i^{i \in I}; P \mid Q] \mid R : \text{pro}[c_h : \wedge \sigma_h]^{h \in H}.$$

Finally, since $J \subseteq H$, we conclude by (Proc Amb) that $E; F \vdash n[c_j^{j \in J}; m[c_i^{i \in I}; P \mid Q] \mid R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Out) By assumption, $m[c_i^{i \in I}; n[c_j^{j \in J}; \text{out } m.P \mid Q] \mid R] \longrightarrow m[c_i^{i \in I}; R] \mid n[c_j^{j \in J}; P \mid Q]$ and $E; F \vdash m[c_i^{i \in I}; n[c_j^{j \in J}; \text{out } m.P \mid Q] \mid R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This judgement must

have been derived as follows:

$$\begin{array}{l}
E; F \vdash m[c_i^{i \in I}; n[c_j^{j \in J}; \text{out } m.P \mid Q \mid R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow E; F \vdash m : \text{amb}[c_l : \wedge \nu_l]^{l \in L} \quad (I \subseteq L) \quad (\text{Exp } n) \quad (\#) \\
\uparrow \dots \\
\uparrow E; c_l : \wedge \nu_l \vdash n[c_j^{j \in J}; \text{out } m.P \mid Q \mid R] : \text{pro}[c_l : \wedge \nu_l]^{l \in L} \quad (\text{Proc Par}) \\
\uparrow E; c_l : \wedge \nu_l \vdash n[c_j^{j \in J}; \text{out } m.P \mid Q] : \text{pro}[c_l : \wedge \nu_l]^{l \in L} \quad (\text{Proc Amb}) \\
\uparrow E; c_l : \wedge \nu_l \vdash n : \text{amb}[c_h : \wedge \sigma_h]^{h \in H} \quad (J \subseteq H) \quad (\text{Exp } n) \quad (\natural) \\
\uparrow \dots \\
\uparrow E; c_h : \wedge \sigma_h \vdash \text{out } m.P \mid Q : \text{pro}[c_h : \wedge \sigma_h]^{h \in H} \quad (\text{Proc Par}) \\
\uparrow E; c_h : \wedge \sigma_h \vdash \text{out } m.P : \text{pro}[c_h : \wedge \sigma_h]^{h \in H} \quad (\text{Proc Action}) \\
\uparrow E; c_h : \wedge \sigma_h \vdash \text{out } m : \text{cap}[c_g : \wedge \sigma_g]^{g \in G} \quad (G \subseteq H) \quad (\text{Exp Out}) \\
\uparrow \dots \\
\uparrow E; c_h : \wedge \sigma_h \vdash P : \text{pro}[c_h : \wedge \sigma_h]^{h \in H} \quad \dots \quad (\dagger) \\
\uparrow \dots \\
\uparrow E; c_h : \wedge \sigma_h \vdash Q : \text{pro}[c_h : \wedge \sigma_h]^{h \in H} \quad \dots \quad (\ddagger) \\
\uparrow \dots \\
\uparrow E; c_l : \wedge \nu_l \vdash R : \text{pro}[c_l : \wedge \nu_l]^{l \in L} \quad \dots \quad (b) \\
\uparrow \dots
\end{array}$$

From (b), (#) and $I \subseteq L$ using (Proc Amb) we have $E; F \vdash m[c_i^{i \in I}; R] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$.

Also, from (†) and (‡) using (Proc Par) we get $E; c_h : \wedge \sigma_h \vdash P \mid Q : \text{pro}[c_h : \wedge \sigma_h]^{h \in H}$.

Then, from (‡) and $J \subseteq H$ using (Proc Amb) we derive $E; c_l : \wedge \nu_l \vdash n[c_j^{j \in J}; P \mid Q] : \text{pro}[c_l : \wedge \nu_l]^{l \in L}$, and by Lemma 3.2.1

$$E; F \vdash n[c_j^{j \in J}; P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$$

which implies using (Proc Par) that $E; F \vdash m[c_i^{i \in I}; R] \mid n[c_j^{j \in J}; P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Open) By assumption, $m[c_i^{i \in I}; \text{open } n.P \mid n[c_j^{j \in J}; R] \mid Q] \longrightarrow m[c_i^{i \in I}; P \mid R \mid Q]$ and

$E; F \vdash m[c_i^{i \in I}; \text{open } n.P \mid n[c_j^{j \in J}; R] \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This judgement must have

been derived as follows:

$$\begin{array}{l}
E; F \vdash m[c_i^{i \in I}; \text{open } n.P \mid n[c_j^{j \in J}; R] \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow \\
E; F \vdash m : \text{amb}[c_l : \wedge v_l]^{l \in L} \quad (I \subseteq L) \quad (\text{Exp } n) \quad (\#) \\
\uparrow \dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{open } n.P \mid n[c_j^{j \in J}; R] \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Par}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{open } n.P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Action}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash \text{open } n : \text{cap}[c_h : \wedge v_h]^{h \in H} \quad (H \subseteq L) \quad (\text{Exp Open}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash n : \text{amb}[c_h : \wedge v_h]^{h \in H} \quad (\text{Exp } n) \\
\uparrow \dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (\dagger) \\
\uparrow \dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash n[c_j^{j \in J}; R] : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Amb}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash n : \text{amb}[c_h : \wedge v_h]^{h \in H} \quad (J \subseteq H) \quad (\text{Exp } n) \\
\uparrow \dots \\
\uparrow \\
E; c_h : \wedge v_h \vdash R : \text{pro}[c_h : \wedge v_h]^{h \in H} \quad \dots \quad (\ddagger) \\
\uparrow \dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (b) \\
\uparrow \dots
\end{array}$$

From (\ddagger) and $H \subseteq L$, it follows by Lemma 3.2.5 that $E; c_l : \wedge v_l \vdash R : \text{pro}[c_l : \wedge v_l]^{l \in L}$.

From the last judgement, together with (\dagger) and (b) , using (Proc Par) we have $E; c_l : \wedge v_l \vdash P \mid R \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L}$. Finally, since $I \subseteq L$ we conclude using the rule (Proc Amb) that $E; F \vdash m[c_i^{i \in I}; P \mid R \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Comm) By assumption, $m[c_i^{i \in I}; c_k \langle M \rangle . R \mid c_k(n : \sigma).P \mid Q] \longrightarrow m[c_i^{i \in I}; R \mid P\{n := M\} \mid Q]$ with $k \in I$, and $E; F \vdash m[c_i^{i \in I}; c_k \langle M \rangle . R \mid c_k(n : \sigma).P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This judgement must have been derived as follows:

$$\begin{array}{l}
E; F \vdash m[c_i^{i \in I}; c_k \langle M \rangle . R \mid c_k(n : \sigma) . P \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow \\
E; F \vdash m : \text{amb}[c_l : \wedge v_l]^{l \in L} \quad (I \subseteq L) \quad (\text{Exp } n) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash c_k \langle M \rangle . R \mid c_k(n : \sigma) . P \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Par}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash c_k \langle M \rangle . R : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Output}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash c_k : \wedge v_k \quad (\text{Exp } c) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash M : v_k \quad \dots \quad (\dagger) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash R : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (\ddagger) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash c_k(n : \sigma) . P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad (\text{Proc Input}) \\
\uparrow \\
E; c_l : \wedge v_l \vdash c_k : \wedge v_k = \wedge \sigma \quad (\text{Exp } c) \\
\uparrow \\
\dots \\
\uparrow \\
E, n : \sigma; c_l : \wedge v_l \vdash P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (b) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash Q : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots \quad (\#) \\
\uparrow \\
\dots
\end{array}$$

From (\dagger) , (b) and Lemma 3.2.6, using the fact that $v_k = \sigma$, we have $E; c_l : \wedge v_l \vdash P\{n := M\} : \text{pro}[c_l : \wedge v_l]^{l \in L}$. Then, by combining this judgement with (\ddagger) and $(\#)$ using (Proc Par) we derive $E; c_l : \wedge v_l \vdash R \mid P\{n := M\} \mid Q : \text{pro}[c_l : \wedge v_l]^{l \in L}$. Finally, since $I \subseteq L$ using (Proc Amb) we conclude that $E; F \vdash m[c_i^{i \in I}; R \mid P\{n := M\} \mid Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Amb) By assumption, $n[c_i^{i \in I}; P] \longrightarrow n[c_i^{i \in I}; Q]$ because $P \longrightarrow Q$ and $E; F \vdash n[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. This judgement must have been derived as follows:

$$\begin{array}{l}
E; F \vdash n[c_i^{i \in I}; P] : \text{pro}[c_k : \wedge \tau_k]^{k \in K} \quad (\text{Proc Amb}) \\
\uparrow \\
E; F \vdash n : \text{amb}[c_l : \wedge v_l]^{l \in L} \quad (I \subseteq L) \quad (\text{Exp } n) \\
\uparrow \\
\dots \\
\uparrow \\
E; c_l : \wedge v_l \vdash P : \text{pro}[c_l : \wedge v_l]^{l \in L} \quad \dots
\end{array}$$

By induction hypothesis $E; c_l : \wedge v_l \vdash Q : \text{pro}[c_l : \wedge v_l]^{l \in L}$, and by (Proc Amb) with $I \subseteq L$, it

follows that $E; F \vdash n[c_i^{i \in I}; Q] : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Res) By assumption, $(\nu n : \sigma).P \longrightarrow (\nu n : \sigma).Q$ because $P \longrightarrow Q$ and $E; F \vdash (\nu n : \sigma).P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Hence, $E, n : \sigma; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ and by induction hypothesis $E, n : \sigma; F \vdash Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Consequently, it follows by (Proc Res) that $E; F \vdash (\nu n : \sigma).Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired.

(Red Par) Immediate using the induction hypotheses.

(Red \equiv) Suppose $P' \longrightarrow Q'$ because $P' \equiv P$ and $P \longrightarrow Q$ and $Q \equiv Q'$. By assumption, $E; F \vdash P' : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. By Lemma 3.2.7 we have $E; F \vdash P : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. By induction hypothesis $E; F \vdash Q : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$. Finally, by Lemma 3.2.7 once again we conclude that $E; F \vdash Q' : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ as desired. \square

Chapter 4

Encoding the Ambient Calculus

The Ambient Calculus can be seen as a special case of **CAC** in which every ambient has a single unnamed channel. In this chapter we present an encoding of the Ambient Calculus into **CAC**, and show that this encoding is semantics preserving and type preserving. For convenience, we refer to the Ambient Calculus in [CG99] simply as **AC**.

4.1 Encoding

Let us begin by extending the set of channel names in **CAC** with the unnamed channel “•”. The mappings $\mathcal{C}^w()$ and $\mathcal{C}()$ are defined next. The former is defined over processes only (as it is only needed at top level) while the latter is defined over expressions, processes, types and type environments.

Definition 4.1.1 (Encoding the Ambient Calculus). Let P be an **AC** process. Define $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. The mapping $\mathcal{C}()$ is defined as follows: for every expression M let $\mathcal{C}(M) = M$, the other cases are considered below.

Processes:

1. $\mathcal{C}(\mathbf{0}) = \mathbf{0}$,
2. $\mathcal{C}(P \mid Q) = \mathcal{C}(P) \mid \mathcal{C}(Q)$,
3. $\mathcal{C}(!P) = !\mathcal{C}(P)$,
4. $\mathcal{C}((\nu n : W).P) = (\nu n : \mathcal{C}(W)).\mathcal{C}(P)$,
5. $\mathcal{C}(M.P) = \mathcal{C}(M).\mathcal{C}(P)$,
6. $\mathcal{C}(M[P]) = \mathcal{C}(M)[\bullet; \mathcal{C}(P)]$,
7. $\mathcal{C}(\langle M \rangle) = \bullet\langle \mathcal{C}(M) \rangle.\mathbf{0}$,
8. $\mathcal{C}((n : W).P) = \bullet(n : \mathcal{C}(W)).\mathcal{C}(P)$.

Types:

Let $\mathcal{C}^?(T)$ be defined as $\mathcal{C}(T)$ if $T \neq \text{Shh}$ and as $\text{amb}[\]$, an arbitrary but fixed **CAC** type, otherwise.

1. $\mathcal{C}(\text{Amb}[T]) = \text{amb}[\bullet : \wedge \mathcal{C}^?(T)]$,
2. $\mathcal{C}(\text{Cap}[T]) = \text{cap}[\bullet : \wedge \mathcal{C}^?(T)]$,
3. $\mathcal{C}(\text{Shh}) = \text{pro}[\]$,
4. $\mathcal{C}(\times(W)) = \text{pro}[\bullet : \wedge \mathcal{C}(W)]$.

For every type environment E in **AC**, let $\mathcal{C}(E)$ be the environment that results by applying $\mathcal{C}()$ to every type in E . □

Lemma 4.1.2 (Properties of $\mathcal{C}()$). *Let P and Q be **AC** processes and M an **AC** expression. Then,*

1. $P \equiv Q$ holds in **AC** if and only if $\mathcal{C}(P) \equiv \mathcal{C}(Q)$ holds in **CAC**,
2. $\mathcal{C}(P\{n := M\}) = \mathcal{C}(P)\{n := M\}$. □

Proof. Part 1 follows by induction on derivations using the definition of \equiv . Part 2 follows by induction on the structure of P using the definition of $\mathcal{C}()$. □

Theorem 4.1.3 (Operational Semantics of **AC).** *Let P and Q be **AC** processes and let w be a name such that $w \notin (\text{fn}(P) \cup \text{bn}(P))$. If $P \longrightarrow Q$ holds in **AC** then $\mathcal{C}^w(P) \longrightarrow \mathcal{C}^w(Q)$ holds in **CAC**.* □

Proof. By induction on derivations using the reduction rules from **AC** as defined in [CG99].

(Red In) We have $P = m[\text{in } n.P' \mid Q'] \mid n[R']$ and $Q = n[m[P' \mid Q'] \mid R']$ in **AC**. It follows by definition of $\mathcal{C}^w()$ and (Red In) in **CAC** that

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; m[\bullet; \text{in } n.\mathcal{C}(P') \mid \mathcal{C}(Q')] \mid n[\bullet; \mathcal{C}(R')]] \\ &\longrightarrow w[\bullet; n[\bullet; m[\bullet; \mathcal{C}(P') \mid \mathcal{C}(Q')] \mid \mathcal{C}(R')]] && \text{(Red Amb)} \\ &= \mathcal{C}^w(Q). \end{aligned}$$

(Red Out) We have $P = m[n[\text{out } m.P' \mid Q'] \mid R']$ and $Q = m[R'] \mid n[P' \mid Q']$ in **AC**. It follows by definition of $\mathcal{C}^w()$ and (Red Out) in **CAC** that

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; m[\bullet; n[\bullet; \text{out } m.\mathcal{C}(P') \mid \mathcal{C}(Q')] \mid \mathcal{C}(R')]] \\ &\longrightarrow w[\bullet; m[\bullet; \mathcal{C}(R') \mid n[\bullet; \mathcal{C}(P') \mid \mathcal{C}(Q')]]] && \text{(Red Amb)} \\ &= \mathcal{C}^w(Q). \end{aligned}$$

(Red Open) We have $P = \text{open } n.P' \mid n[Q']$ and $Q = P' \mid Q'$ in **AC**. It follows by definition of $\mathcal{C}^w()$ and (Red Open) in **CAC** that

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; \text{open } n.\mathcal{C}(P') \mid n[\bullet; \mathcal{C}(Q')]] \\ &\longrightarrow w[\bullet; \mathcal{C}(P') \mid \mathcal{C}(Q')] && \text{(Red Amb)} \\ &= \mathcal{C}^w(Q). \end{aligned}$$

(Red Comm) We have $P = (n : W).P' \mid \langle M \rangle$ and $Q = P'\{n := M\}$ in **AC**. It follows by definition of $\mathcal{C}^w()$ and (Red Comm) in **CAC** that

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; \bullet(n : \mathcal{C}(W)).\mathcal{C}(P') \mid \bullet\langle M \rangle.\mathbf{0}] \\ &\longrightarrow w[\bullet; \mathcal{C}(P')\{n := M\}] && \text{(Red Amb)} \\ &= w[\bullet; \mathcal{C}(P'\{n := M\})] && \text{(By Lemma 4.1.2)} \\ &= \mathcal{C}^w(Q). \end{aligned}$$

(Red Res) We have $P = (\nu n : W).P'$ and $Q = (\nu n : W).Q'$ with $P' \longrightarrow Q'$. By induction hypothesis $\mathcal{C}^w(P') \longrightarrow \mathcal{C}^w(Q')$ or, in expanded form, $w[\bullet; \mathcal{C}(P')] \longrightarrow w[\bullet; \mathcal{C}(Q')]$. It follows that $\mathcal{C}(P') \longrightarrow \mathcal{C}(Q')$ and by (Red Res) in **CAC** we have $(\nu n : \mathcal{C}(W)).\mathcal{C}(P') \longrightarrow$

$(\nu n : \mathcal{C}(W)).\mathcal{C}(Q')$. Hence,

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; (\nu n : \mathcal{C}(W)).\mathcal{C}(P')] \\ &\longrightarrow w[\bullet; (\nu n : \mathcal{C}(W)).\mathcal{C}(Q')] \\ &= \mathcal{C}^w(Q). \end{aligned} \quad (\text{Red Amb})$$

(Red Amb) We have $P = n[P']$ and $Q = n[Q']$ with $P' \longrightarrow Q'$. By induction hypothesis

$\mathcal{C}^w(P') \longrightarrow \mathcal{C}^w(Q')$ or, in expanded form, $w[\bullet; \mathcal{C}(P')] \longrightarrow w[\bullet; \mathcal{C}(Q')]$. It follows that

$\mathcal{C}(P') \longrightarrow \mathcal{C}(Q')$ and by (Red Amb) in **CAC** we have $n[\bullet; \mathcal{C}(P')] \longrightarrow n[\bullet; \mathcal{C}(Q')]$. Hence,

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; n[\bullet; \mathcal{C}(P')]] \\ &\longrightarrow w[\bullet; n[\bullet; \mathcal{C}(Q')]] \\ &= \mathcal{C}^w(Q). \end{aligned} \quad (\text{Red Amb})$$

(Red Par) We have $P = P' \mid R'$ and $Q = Q' \mid R'$ with $P' \longrightarrow Q'$. By induction hypothesis

$\mathcal{C}^w(P') \longrightarrow \mathcal{C}^w(Q')$ or, in expanded form, $w[\bullet; \mathcal{C}(P')] \longrightarrow w[\bullet; \mathcal{C}(Q')]$. It follows that

$\mathcal{C}(P') \longrightarrow \mathcal{C}(Q')$ and by (Red Par) in **CAC** we have $\mathcal{C}(P') \mid \mathcal{C}(R') \longrightarrow \mathcal{C}(Q') \mid \mathcal{C}(R')$.

Hence,

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; \mathcal{C}(P') \mid \mathcal{C}(R')] \\ &\longrightarrow w[\bullet; \mathcal{C}(Q') \mid \mathcal{C}(R')] \\ &= \mathcal{C}^w(Q). \end{aligned} \quad (\text{Red Amb})$$

(Red \equiv) We have $P \equiv P'$ and $P' \longrightarrow Q'$ and also $Q' \equiv Q$. By induction hypothesis $\mathcal{C}^w(P') \longrightarrow$

$\mathcal{C}^w(Q')$ or, in expanded form, $w[\bullet; \mathcal{C}(P')] \longrightarrow w[\bullet; \mathcal{C}(Q')]$. It follows that $\mathcal{C}(P') \longrightarrow$

$\mathcal{C}(Q')$ and by (Red \equiv) in **CAC** and Lemma 4.1.2 we have $\mathcal{C}(P) \longrightarrow \mathcal{C}(Q)$. Hence,

$$\begin{aligned} \mathcal{C}^w(P) &= w[\bullet; \mathcal{C}(P)] \\ &\longrightarrow w[\bullet; \mathcal{C}(Q)] \\ &= \mathcal{C}^w(Q). \end{aligned} \quad (\text{Red Amb})$$

□

Lemma 4.1.4 (Encoding of Environments). *Let E be a type environment in **AC**. If $E \vdash \diamond$ then $\mathcal{C}(E); \emptyset \vdash \diamond$.* □

Proof. By induction on the length of E . The proof is trivial as the premises of the rule (Env n) are identical in both systems. □

Lemma 4.1.5 (Typing of Expressions in AC). *Let M be an **AC** expression. If $E \vdash M : W$ is derivable in **AC** then $\mathcal{C}(E); \emptyset \vdash \mathcal{C}(M) : \mathcal{C}(W)$ is derivable in **CAC**.* □

Proof. By induction on derivations using Definition 4.1.1. □

Theorem 4.1.6 (Typing of Processes in AC). *Let R be an **AC** process and let w be a name such that $w \notin (fn(R) \cup bn(R))$. If $E \vdash R : T$ is derivable in **AC** and $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \diamond$ then $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(R) : \mathcal{C}(\text{Shh})$ is derivable in **CAC**.* □

Proof. By induction on derivations.

(Proc Zero) We have $E \vdash \mathbf{0} : T$ because $E \vdash \diamond$. Suppose $T = \text{Shh}$. It follows by Lemma 4.1.4 and (Proc Zero) in **CAC** that $\mathcal{C}(E); \emptyset \vdash \mathcal{C}(\mathbf{0}) : \mathcal{C}(\text{Shh})$, since $\mathcal{C}(\mathbf{0}) = \mathbf{0}$ and $\mathcal{C}(\text{Shh}) = \text{pro}[\]$. By Lemma 3.2.3 (twice) we get

$$\mathcal{C}(E), w : \text{amb}[\bullet : \wedge \text{amb}[\]]; \bullet : \wedge \text{amb}[\] \vdash \mathcal{C}(\mathbf{0}) : \mathcal{C}(\text{Shh}).$$

By (Exp n) we have $\mathcal{C}(E), w : \text{amb}[\bullet : \wedge \text{amb}[\]]; \emptyset \vdash w : \text{amb}[\bullet : \wedge \text{amb}[\]]$. Therefore, by (Proc Amb) we derive $\mathcal{C}(E), w : \text{amb}[\bullet : \wedge \text{amb}[\]]; \emptyset \vdash w[\bullet; \mathcal{C}(\mathbf{0})] : \mathcal{C}(\text{Shh})$ as desired. The case for $T \neq \text{Shh}$ is analogous.

(Proc Par) We have $E \vdash P \mid Q : T$ because $E \vdash P : T$ and $E \vdash Q : T$. By induction hypothesis we get $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ and $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(Q) : \mathcal{C}(\text{Shh})$ with $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$ and $\mathcal{C}^w(Q) = w[\bullet; \mathcal{C}(Q)]$. It follows that the judgements

$$\begin{aligned} & \mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)], \\ & \mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(Q) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)] \end{aligned}$$

are derivable. Hence, by (Proc Par) and (Proc Amb) in **CAC** we get $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash w[\bullet; \mathcal{C}(P) \mid \mathcal{C}(Q)] : \mathcal{C}(\text{Shh})$ as desired.

(Proc Res) We have $E \vdash (\nu n : \text{Amb}[T]).P : S$ because $E, n : \text{Amb}[T] \vdash P : S$. By induction hypothesis we get $\mathcal{C}(E, n : \text{Amb}[T], w : \text{Amb}[S]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ where $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. It follows that $\mathcal{C}(E, n : \text{Amb}[T], w : \text{Amb}[S]); \bullet : \wedge \mathcal{C}^?(S) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(S)]$. By Lemma 3.2.2 and the rule (Proc Res) we derive

$$\mathcal{C}(E, w : \text{Amb}[S]); \bullet : \wedge \mathcal{C}^?(S) \vdash (\nu n : \mathcal{C}(\text{Amb}[T])).\mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(S)].$$

Hence, by (Proc Amb) in **CAC**, $\mathcal{C}(E, w : \text{Amb}[S]); \emptyset \vdash w[\bullet; (\nu n : \mathcal{C}(\text{Amb}[T])).\mathcal{C}(P)] : \mathcal{C}(\text{Shh})$ as desired.

(Proc Repl) We have $E \vdash !P : T$ because $E \vdash P : T$. By induction hypothesis, $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ where $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. It follows that $\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)]$. By (Proc Repl) in **CAC** we derive the judgement

$$\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash !\mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)].$$

Hence, by (Proc Amb) in **CAC** we conclude that $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash w[\bullet; !\mathcal{C}(P)] : \mathcal{C}(\text{Shh})$ as desired.

(Proc Action) We have $E \vdash M.P : T$ because $E \vdash M : \text{Cap}[T]$ and $E \vdash P : T$. By induction hypothesis we get $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ with $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. It follows that $\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)]$. By Lemma 4.1.5 we get

$\mathcal{C}(E); \emptyset \vdash \mathcal{C}(M) : \mathcal{C}(\text{Cap}[T])$, and by Lemma 3.2.3 $\mathcal{C}(E); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(M) : \mathcal{C}(\text{Cap}[T])$.

Therefore, by (Proc Action) in **CAC** we have

$$\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(M).\mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)].$$

Finally, by (Proc Amb) we conclude $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash w[\bullet; \mathcal{C}(M).\mathcal{C}(P)] : \mathcal{C}(\text{Shh})$ as desired.

(Proc Amb) We have $E \vdash M[P] : S$ because $E \vdash M : \text{Amb}[T]$ and $E \vdash P : T$. By induction hypothesis we get $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ with $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. It follows that $\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(T) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}^?(T)]$. By Lemma 4.1.5 we get $\mathcal{C}(E); \emptyset \vdash \mathcal{C}(M) : \mathcal{C}(\text{Amb}[T])$, and by Lemma 3.2.3 $\mathcal{C}(E); \bullet : \wedge \mathcal{C}^?(S) \vdash \mathcal{C}(M) : \mathcal{C}(\text{Amb}[T])$. Therefore, by Lemma 3.2.2 and the rule (Proc Amb) in **CAC** we have

$$\mathcal{C}(E, w : \text{Amb}[T]); \bullet : \wedge \mathcal{C}^?(S) \vdash \mathcal{C}(M)[\bullet; \mathcal{C}(P)] : \text{pro}[\bullet : \wedge \mathcal{C}^?(S)].$$

Finally, by (Proc Amb) once again we conclude $\mathcal{C}(E, w : \text{Amb}[T]); \emptyset \vdash w[\bullet; \mathcal{C}(M)[\bullet; \mathcal{C}(P)]] : \mathcal{C}(\text{Shh})$ as desired.

(Proc Input) We have $E \vdash (n : W).P : T$ because $E, n : W \vdash P : T$ where $T = \times(W)$. Suppose $W \neq \text{Shh}$. By induction hypothesis we get $\mathcal{C}(E, n : W, w : \text{Amb}[W]); \emptyset \vdash \mathcal{C}^w(P) : \mathcal{C}(\text{Shh})$ with $\mathcal{C}^w(P) = w[\bullet; \mathcal{C}(P)]$. It follows that $\mathcal{C}(E, n : W, w : \text{Amb}[W]); \bullet : \wedge \mathcal{C}(W) \vdash \mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}(W)]$, and by (Exp e) that $\mathcal{C}(E, w : \text{Amb}[W]); \bullet : \wedge \mathcal{C}(W) \vdash \bullet : \wedge \mathcal{C}(W)$. Therefore, by Lemma 3.2.2 and the rule (Proc Input) in **CAC** we have

$$\mathcal{C}(E, w : \text{Amb}[W]); \bullet : \wedge \mathcal{C}(W) \vdash \bullet(n : \mathcal{C}(W)).\mathcal{C}(P) : \text{pro}[\bullet : \wedge \mathcal{C}(W)].$$

Finally, by (Proc Amb) we conclude $\mathcal{C}(E, w : \text{Amb}[W]); \emptyset \vdash w[\bullet; \bullet(n : \mathcal{C}(W)).\mathcal{C}(P)] : \mathcal{C}(\text{Shh})$

as desired. The case for $W = \text{Shh}$ is analogous.

(Proc Output) We have $E \vdash \langle M \rangle : T$ because $E \vdash M : W$ where $T = \times(W)$. Suppose $W \neq \text{Shh}$.

By Lemma 4.1.5 we have $\mathcal{C}(E); \emptyset \vdash \mathcal{C}(M) : \mathcal{C}(W)$, and then by Lemma 3.2.3 we get $\mathcal{C}(E); \bullet : \wedge \mathcal{C}(W) \vdash \mathcal{C}(M) : \mathcal{C}(W)$. Moreover, by (Exp c) and (Proc Zero), respectively, we derive $\mathcal{C}(E); \bullet : \wedge \mathcal{C}(W) \vdash \bullet : \wedge \mathcal{C}(W)$ and $\mathcal{C}(E); \bullet : \wedge \mathcal{C}(W) \vdash \mathbf{0} : \text{pro}[\bullet : \wedge \mathcal{C}(W)]$. Then, it follows from the last three judgements that

$$\mathcal{C}(E); \bullet : \wedge \mathcal{C}(W) \vdash \bullet \langle \mathcal{C}(M) \rangle . \mathbf{0} : \text{pro}[\bullet : \wedge \mathcal{C}(W)].$$

Finally, by (Proc Amb) we conclude $\mathcal{C}(E); \emptyset \vdash w[\bullet; \bullet \langle \mathcal{C}(M) \rangle . \mathbf{0}] : \mathcal{C}(\text{Shh})$ as desired. The case for $W = \text{Shh}$ is analogous. □

Chapter 5

Conclusions to Part I

In this part we presented a process calculus that we called Channel Ambient Calculus or, more concisely, **CAC**. This process calculus combines design principles from the Ambient Calculus and from the π -Calculus. From the former, it inherits mobility primitives and a notion of location, from the latter, the ability of processes to communicate over named channels instead of over the ether.

We also devised a type system for our calculus, which we showed to be sound by proving a Structural Congruence theorem and Subject Reduction theorem. In Chapter 4 we outlined an encoding of the Ambient Calculus into **CAC**: the simplicity of this encoding is a clear indication that **CAC** is, as advertised, a natural extension of the Ambient Calculus.

5.1 Discussion and Future Work

In Chapter 3 we presented a provably sound type system for **CAC**. In the context of that type system, an ambient that is opened inside a parent ambient can only use channels that are avail-

able in the latter. This restriction is enforced by the side conditions in the rules (Proc Action) and (Exp Action). In fact, these side conditions can be regarded as a form of *inlined subtyping*, where the set of channels in a capability type is allowed to grow to match that of a process type (in (Proc Action)) or a capability type (in (Exp Action)). Thus, an alternative way to set up a type system for **CAC** is by using a subsumption rule together with the subtyping relation defined by the following rules:

$$\begin{array}{c}
 \text{(SubChaset)} \\
 \frac{I \subseteq J}{\vdash c_i : \wedge \tau_i^{i \in I} \leq c_j : \wedge \tau_j^{j \in J}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(SubCap)} \\
 \frac{\vdash \varsigma \leq \varsigma'}{\vdash \text{cap}[\varsigma] \leq \text{cap}[\varsigma']}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(SubPro)} \\
 \frac{\vdash \varsigma \leq \varsigma'}{\vdash \text{pro}[\varsigma] \leq \text{pro}[\varsigma']}
 \end{array}$$

where, for convenience, we think of a channel set $c_i : \wedge \tau_i^{i \in I}$ as type on its own, as opposed to a subpart of a capability or process type. This type system can be shown to be as powerful as the one presented in Chapter 3.

More interestingly, however, is to consider a type system in which subtyping can take place not only in *width* (as in the system shown above) but also in *depth*. The main difficulty in defining such a system is that channels can be used for input and output and, as a result, they can only be subtyped *invariantly* (i.e., neither *covariantly* nor *contravariantly*). At an abstract level, a channel is similar to a reference cell or an (updatable) method in an object: a reference cell can be read and written, a method can be updated or selected, and a channel can be used for input and output. A number of solutions have been proposed to overcome this problem, for example, the use of Split Types in [BPG00a, BPG00b].

The use of input only channels and output only channels is another solution proposed in the context of the π -Calculus, and the one we elaborate next. Clearly, if a channel can be used for input and output it can also be used just for input or just for output. Let us define $?\tau$ and $!\tau$ as

the types of input only channels and output only channels, respectively. Subtyping over channel types, capability types and process types can, therefore, be defined as follows:

$$\begin{array}{c}
\text{(SubIOCha)} \\
\frac{}{\vdash \wedge_{\tau} \leq \wedge_{\tau}}
\end{array}
\qquad
\begin{array}{c}
\text{(SubOCha)} \\
\frac{\vdash \tau' \leq \tau \quad \alpha \in \{\wedge, !\}}{\vdash \alpha_{\tau} \leq !\tau'}
\end{array}
\qquad
\begin{array}{c}
\text{(SubICha)} \\
\frac{\vdash \tau \leq \tau' \quad \alpha \in \{\wedge, ?\}}{\vdash \alpha_{\tau} \leq ?\tau'}
\end{array}$$

$$\begin{array}{c}
\text{(SubChaSet)} \\
\frac{\vdash \beta_i \sigma_i \leq \alpha_i \tau_i \quad \forall i \in I \quad I \subseteq J}{\vdash c_i : \alpha_i \tau_i^{i \in I} \leq c_j : \beta_j \sigma_j^{j \in J}}
\end{array}
\qquad
\begin{array}{c}
\text{(SubCap)} \\
\frac{\vdash \varsigma \leq \varsigma'}{\vdash \text{cap}[\varsigma] \leq \text{cap}[\varsigma']}
\end{array}
\qquad
\begin{array}{c}
\text{(SubPro)} \\
\frac{\vdash \varsigma \leq \varsigma'}{\vdash \text{pro}[\varsigma] \leq \text{pro}[\varsigma']}
\end{array}$$

Subtyping over ambient types, the types of ambient names, is unfortunately also invariant. An ambient name can be used in two contexts: (i) to open an ambient (read) and (ii) to construct an ambient (write). A solution to this problem, which involves splitting the type of an ambient as done in [BPG00a] for objects, has been explored in [Zim00] and in [AKPG01].

The complete development of the type system described in the previous paragraphs, which clearly extends that presented in Chapter 3, is left as future work.

Part II

XML

Chapter 6

Introduction to Part II

6.1 XML

XML is a simple and portable way of representing structured data in the form of trees. Structured data consists of *content* (the data to be represented) and *markup* (additional data describing the role played by the content). For example, an e-mail message represented as an XML document may look like this:

```
<email>
  <sender>santiago@cs.bu.edu</sender>
  <receiver>all@cs.bu.edu</receiver>
  <subject>An e-mail message as an XML document</subject>
  <message>This is an e-mail message.</message>
</email>
```

The content of this document includes *strings* such as ‘santiago@cs.bu.edu’ or ‘This is an e-mail message.’, while the markup is defined using *tags* such as <sender> or <message>; closing tags such as </sender> or </message> are needed to delimit the scope of their corresponding opening tags.

Every XML document can be represented as a tree that includes a *root* node (not explicitly

shown in the document), an interior node for every opening tag present in the document and an exterior node (i.e. a leaf) for every string that is part of the content.¹ The children of an interior node are the (interior or exterior) nodes delimited by its opening and closing tags.

An *XML Processor* is a program that reads XML documents and provides access to their content and structure. This definition is intentionally “broad”, as it includes programs such as XML parsers, XML transformers and XML query systems, among others.

XML is an ideal language for machine-to-machine (or peer-to-peer) exchange of data because (i) it is portable, so there is no need to worry about things like integer representations (big endian vs. little endian) or string representations (null terminated vs. fixed length) and (ii) structured data is easily represented in the form of trees. Interestingly, the ability of XML to represent (most) structured data is often an important weakness at the same time. Consider an application that is expecting an e-mail in the form of an XML document as the one shown above:

- Does it expect the initial tag to be `<email>` or `<e-mail>`?
- Does it expect the receiver’s data to precede or to follow the sender’s data?
- Will it look for the `subject` tag as a child or as a sibling of the `message` tag?

The problems posed by these questions can be addressed by the use of a *type system*. A type system is needed in order to keep track of the meta-data (i.e. data about the data) which can be used by a consumer application to find/extract what it needs from an XML document.

A number of type systems for XML have been proposed (an are still being proposed!), among them, Document Type Declarations or DTDs [Wora], XML Schemas [Wore] and Regular Express-

¹For our purposes, “whitespace” nodes sitting between markup nodes are ignored. As a result, the same tree is denoted by many (in fact, infinitely many!) XML documents. Notice, on the other hand, that whitespace inside content nodes is very important as it may change the intended meaning of the data stored in that node.

sion Types [HP00, HP01]. At the time of writing the most widely used type system is DTDs, mainly because it was the first type system proposed and because it is simple to write tools that support it. Despite the variety of type systems, their use —with the exception of Regular Expression Types in XDuce— is still rather limited in a sense to be made precise next.

Returning to our running example, suppose we call the producer (or sender) of an e-mail message APP A and the consumer (or receiver) of that message APP B. In order to ensure that APP B “understands” the message sent by APP A, we place a *type checker* (i.e. a program that checks if an XML document validates a given XML type) in the communication path between the two applications. The main disadvantages of this architecture are the following:

1. Inefficiency: every XML document produced by APP A must be validated, which may be very time consuming specially if APP A has been very carefully written so that the documents it outputs *always* validate the type in question.
2. Error Recovery: it is unclear what to do with a document produced by APP A that does not pass the type checker. Should it be thrown away? Should it be handed to APP B?
3. Incorrect Implementation: it is possible for APP A (resp. APP B) to produce (resp. consume) documents of a type different from that of its specification. Stated differently, both APP A and APP B must internally type check as well.

Mainly due to (1), these type checkers are often not employed in practice, leaving the door open to run-time errors that can occur as a result of incorrectly written applications.

The use of *static typing* as opposed to *dynamic typing* is an alternative to solving the problems listed above. If we can determine statically (i.e. at compile time) that both applications will produce and consume documents of the same type, then there is no need for dynamic checking.

In the next section we present an example written in the syntax of the $\delta\lambda$ -Calculus (see Chapter 7). As suggested by its name, the $\delta\lambda$ -Calculus is a λ -Calculus [Chu41, Bar84] extended with primitives for manipulating documents (hence the use of the Greek letter δ).

6.2 Motivating Example

The $\delta\lambda$ -Calculus is the first (to the best of our knowledge) typed calculus for writing XML processors. It is similar in spirit to the language presented in [HP00, HP01] in the use of a static type system, but at the same time different as it lacks pattern matching and many other constructs that are often present in a full-blown language and not in a calculus.

Our main objective is to integrate the $\delta\lambda$ -Calculus (or at least a subset of it) with the Channelled Ambient Calculus introduced in Part I, so that mobile agents can exchange data in the form of XML documents in a typed-safe manner.

In what follows, we use a more compact notation when writing XML documents. To exemplify, the e-mail message of the Section 6.1 is written as:

```
email<sender<'santiago@cs.bu.edu'>,
      receiver<'all@cs.bu.edu'>,
      subject<'An e-mail message as an XML document'>,
      message<'This is an e-mail message.'>>
```

That is, we omit closing tags and use “<” and “>” to define the scope (i.e. the children) of a markup element; use “,” to separate sibling nodes and single quotes to delimit strings that are part of the content. In preparation for a more rigorous study of XML, we omit important (yet not fundamental for our purposes) features of XML. Specifically, we exclude from our formal presentation things like attributes, namespaces, references, processing instructions, etc.; and concentrate exclusively on what we defined as *content* and *markup*.

A typical use of an XML Processor is to transform an XML document into an HTML document for displaying purposes [Worb, Word]. The following $\delta\lambda$ -Calculus procedure defines such a transformation for our e-mail messages represented as XML documents:

```

 $\lambda x : \tau.$ html<head<h3<'Subject:',  $x$ /email.3/subject>>,
      body<p<'Sender:',  $x$ /email.1/sender>,
          p<'Receiver:',  $x$ /email.2/receiver>,
          p<'Message:',  $x$ /email.4/message>>>

```

The expression x /email selects *all* the children of the interior node labeled email. As a result, we need to use a “projection” operator to select a specific one: the expression x /email.1 selects the first (leftmost) children of email, the expression x /email.2 select the second, and so on. In this example, the type τ (which annotates the bound occurrence of the variable x) is similar in structure to the XML document itself. That is,

$$\tau = \text{email}\langle\text{sender}\langle\text{string}\rangle, \text{receiver}\langle\text{string}\rangle, \text{subject}\langle\text{string}\rangle, \text{message}\langle\text{string}\rangle\rangle$$

where string is a primitive (or ground) type.

The transformation shown above would have been more interesting had we allowed multiple receivers of an e-mail message. In this case, an appropriate type τ' would be defined as follows:

$$\tau' = \text{email}\langle\text{sender}\langle\text{string}\rangle, \text{receiver}\langle\text{string}\rangle+, \text{subject}\langle\text{string}\rangle, \text{message}\langle\text{string}\rangle\rangle$$

where + is the familiar operator used to define a sequence of length at least one. In the following chapters, we present examples that show how to operate on document whose types include regular operators such as + or *. At this point, it suffices to say that “unbounded” sequences (such as those denoted by types like receiver<string>+) can be manipulated using standard operators like *car* and *cdr* and the use of recursion.

Chapter 7

Untyped $\delta\lambda$ -Calculus

7.1 Untyped δ -Calculus

The $\delta\lambda$ -Calculus is defined as the union of two separate calculi: the δ -Calculus, with primitives to manipulate XML documents; and the λ -Calculus [Chu41, Bar84], with primitives to define abstractions over XML documents. We begin by introducing the syntax of the δ -Calculus, we will later extend this syntax with the λ -Calculus primitives, namely, variables, abstractions and applications.

Let ℓ range over an infinite set of labels Lab . We use labels drawn from Lab to define the *markup* of a document. Let s range over a set of strings $\text{Str} = \{w \mid w \in \Sigma^*\}$ where Σ is an alphabet (e.g. Ascii or Unicode) that we leave unspecified as it does not affect the analysis that follows. We use strings drawn from Str to define the *content* of a document. The syntax of the δ -Calculus is shown below.

Constructors for document expressions include strings s , lists of expressions obtained from $[]$ and $D :: D'$, fixed-length tuples of expressions D_1, \dots, D_n , with $n \geq 2$, and labeled expressions

$\ell \langle D \rangle$. Additionally, there are destructors for lists ($D.\text{hd}$ and $D.\text{tl}$), for tuples ($D.n$) and for labeled expressions (D/ℓ). A list of expressions like $[D_1, \dots, D_n]$ is syntactic sugar for $D_1 :: \dots (D_n :: [])$

$D, D' \in \text{DExp}$	$::=$	s	(String)
		$[]$	(Nil)
		$D :: D'$	(Cons)
		$D_1, \dots, D_n \quad (n \geq 2)$	(Tuple)
		$\ell \langle D \rangle$	(Label)
		$D.\text{hd}$	(Head)
		$D.\text{tl}$	(Tail)
		$D.n$	(Projection)
		D/ℓ	(Selection)
		\dots	

We assume that “.” and “/” have higher precedence than “::”, which in turn, has higher precedence than “,”; operators “.” and “/” associate to the left and operator “::” associates to the right. For example, the expression $D_1 :: D_2/\ell, D_3.\text{hd}$ is equivalent to $(D_1 :: (D_2/\ell)), (D_3.\text{hd})$.

We provide constructors for defining *bounded* sequences (or tuples) as well as *unbounded* sequences (or lists). This is an important difference between the representation of documents in the δ -Calculus and in XML. In the latter, there is no syntactic difference between those two kinds of sequences: this information can only be recovered by analyzing the document’s DTD. That is, by looking at an XML document, there is not way of telling whether a subtree is intended to be repeated or not.

To exemplify, let us consider the e-mail message from Chapter 6:

```
<email>
  <sender>santiago@cs.bu.edu</sender>
  <receiver>all@cs.bu.edu</receiver>
  <subject>An e-mail message as an XML document</subject>
  <message>This is an e-mail message.</message>
</email>
```

Given this document there is no way of telling, for example, if multiple receivers of a message are allowed. In other words, there is no information in the document as to whether 2 or 15 or an unbounded number of receivers can be specified.

In the δ -Calculus, we require the programmer to be more specific, and use the appropriate constructors depending on whether a subtree is intended to be repeated or not. For example, the e-mail message shown above, under the assumption that multiple receivers are allowed, would correspond to the following δ -Calculus expression:

```
email<sender<'santiago@cs.bu.edu'>,
      [receiver<'all@cs.bu.edu'>],
      subject<'An e-mail message as an XML document'>,
      message<'This is an e-mail message.'>>
```

The operational semantics of the δ -Calculus is presented in Figure 7.1. Let V range over a set of values. This set includes strings, fully-evaluated lists and fully-evaluated tuples. More precisely,

$V, V' \in DVal$	$::=$	s	(String)
		$[]$	(Nil)
		$V :: V'$	(Cons)
		V_1, \dots, V_n	(Tuple)
		$\ell \langle V \rangle$	(Label)
		\dots	

Notice that, by construction, every XML document is a value and every value is an XML document: a δ -Calculus expression that does not get stuck must evaluate to an XML document. From an XML document we can obtain a δ -Calculus value by simply inserting the appropriate list delimiters—we shall return to this issue in Chapter 9 when we present a transformation from XML documents into δ -Calculus expressions. From a δ -Calculus value we can get an XML document by simply ignoring all list delimiters.

$(V::V')\cdot\text{hd} \longrightarrow V$	(Red Head)
$(V::V')\cdot\text{tl} \longrightarrow V'$	(Red Tail)
$(V_1, \dots, V_n)\cdot k \longrightarrow V_k \quad k \in 1..n$	(Red Proj)
$(\ell\langle V \rangle)/\ell \longrightarrow V$	(Red Sel)
$D \longrightarrow D' \Rightarrow D\cdot\text{hd} \longrightarrow D'\cdot\text{hd}$	(Red DHead)
$D \longrightarrow D' \Rightarrow D\cdot\text{tl} \longrightarrow D'\cdot\text{tl}$	(Red DTail)
$D \longrightarrow D' \Rightarrow D\cdot k \longrightarrow D'\cdot k$	(Red DProj)
$D \longrightarrow D' \Rightarrow D/\ell \longrightarrow D'/\ell$	(Red DSel)
$D \longrightarrow D' \Rightarrow D::D'' \longrightarrow D'::D''$	(Red ConsHead)
$D \longrightarrow D' \Rightarrow V::D \longrightarrow V::D'$	(Red ConsTail)
$D \longrightarrow D' \Rightarrow \ell\langle D \rangle \longrightarrow \ell\langle D' \rangle$	(Red Label)
$D_k \longrightarrow D'_k \Rightarrow V_1, \dots, V_{k-1}, D_k, \dots, D_n \longrightarrow V_1, \dots, V_{k-1}, D'_k, \dots, D_n$	(Red Tuple)

Figure 7.1. Operational Semantics: δ -Calculus.

The first four rules in Figure 7.1 are the axioms of the reduction relation \longrightarrow . These axioms are applicable only if the document expression to the left of the operator is fully evaluated. Accordingly, the next four rules state that evaluation is permitted to the left of “.” and “/”, while the rest of the rules define the way tuples and lists are evaluated. Due to the lack of side effects in our calculus, there is nothing special about the left-to-right evaluation of tuples and lists; other evaluation strategies are equally plausible.

The rule (Red Sel) selects the value V from the expression $\ell\langle V \rangle$. Hence, there is no way of selecting a labeled value from either a tuple or a list. For example, if $D = (\ell_1\langle V_1 \rangle, \ell_2\langle V_2 \rangle)$ then in order to obtain the value V_1 we must write $D\cdot 1/\ell_1$.

EXAMPLE 7.1.1. Suppose we are interested in obtaining the e-mail address of the first receiver from a message represented as an XML document (i.e. a value in our calculus). Let D_i for $i \in 1..3$ be defined as follows:

$$\begin{aligned}
D_1 &= \text{email}\langle D_2 \rangle \\
D_2 &= \text{sender}\langle \text{'santiago@cs.bu.edu'} \rangle, D_3, \\
&\quad \text{subject}\langle \text{'An e-mail message as an XML document'} \rangle, \\
&\quad \text{message}\langle \text{'This is an e-mail message.'} \rangle \\
D_3 &= [\text{receiver}\langle \text{'all@cs.bu.edu'} \rangle, \text{receiver}\langle \text{'santiago@cs.bu.edu'} \rangle]
\end{aligned}$$

The following reduction sequence shows that $((D_1/\text{email}).2).\text{hd}/\text{receiver}$ is the appropriate δ -Calculus expression for our purpose.

$$\begin{array}{ll}
((D_1/\text{email}).2).\text{hd}/\text{receiver} & \text{(Red DSel)} \\
\longrightarrow ((D_2.2).\text{hd})/\text{receiver} & \text{(Red DSel)} \\
\longrightarrow (D_3.\text{hd})/\text{receiver} & \text{(Red DSel)} \\
\longrightarrow (\text{receiver}\langle \text{'all@cs.bu.edu'} \rangle)/\text{receiver} & \text{(Red Sel)} \\
\longrightarrow \text{'all@cs.bu.edu'} &
\end{array}$$

Because all the D_i 's are fully-evaluated, the expression shown above simply traverses the document to select the desired subtree. □

7.2 Untyped $\delta\lambda$ -Calculus

For the purpose of reasoning about transformations over XML documents, we need a way of writing *document abstractions* so that, for example, we can map XML documents that share a similar structure into HTML (or XHTML) documents for displaying purposes (cf. Chapter 6). For this reason, we extend the syntax of the δ -Calculus with primitives from the λ -Calculus [Chu41, Bar84].

$D, D' \in \text{DExp} ::= \dots$	δ -Calculus
x	(Variable)
$\lambda x.D$	(Abstraction)
DD'	(Application)
\dots	

$(\lambda x.D)V \longrightarrow D\{x := V\}$	(Red Beta)
$D \longrightarrow D' \Rightarrow DD'' \longrightarrow D'D''$	(Red AppLeft)
$D \longrightarrow D' \Rightarrow VD \longrightarrow VD'$	(Red AppRight)

Figure 7.2. Operational Semantics: CBV λ -Calculus.

Additionally, we extend the operational semantics with rules corresponding to the call-by-value λ -Calculus (cf. Figure 7.2), and the set of values with variables (x) and abstractions ($\lambda x.D$).

Example 7.1.1 from Section 7.1 can now be re-written using constructs from the λ -Calculus fragment. That is, we can write an abstraction that expects e-mail messages in the form of XML documents and returns the e-mail address of the first receiver.

EXAMPLE 7.2.1. Let D_1 be defined as in Example 7.1.1, and let D be the document abstraction $(\lambda x.x/\text{email.2.hd/receiver})$. Then, it is immediate to show that the application (DD_1) is such that $(DD_1) \longrightarrow \text{'all@cs.bu.edu'}$. □

In order to process XML documents containing unbounded tuples (a.k.a. lists), we need our calculus to be powerful enough to express *unbounded computations*. Even though, in principle, this is possible using standard encodings and the λ -Calculus, in preparation to add a static type system and foreseeing the untypability of some of those encodings (e.g. a fix-point operator such as $Y = \lambda f.(\lambda x.f(\lambda z.(xx)z))(\lambda x.f(\lambda z.(xx)z))$), we extend the syntax with a few additional primitives. These primitives include conditional expressions, recursive definitions (fix), predicates and boolean values (needed for the evaluation of conditionals). For convenience, we also include an operator \oplus for concatenating strings that we use in the examples that follow.

The extended syntax is shown below. Among these new primitives, only true and false are values; the reduction rules for the other constructs is shown in Figure 7.3. In (Red Concat), the

$\text{fix } x.D \longrightarrow D\{x := \text{fix } x.D\}$	(Red Fix)
$V \oplus V' \longrightarrow V'' \quad (V'' = \text{concat}(V, V'))$	(Red Concat)
$D \longrightarrow D' \Rightarrow D \oplus D'' \longrightarrow D' \oplus D''$	(Red ConcatLeft)
$D \longrightarrow D' \Rightarrow V \oplus D \longrightarrow V \oplus D'$	(Red ConcatRight)
$\text{if true then } D_1 \text{ else } D_2 \longrightarrow D_1$	(Red IfTrue)
$\text{if false then } D_1 \text{ else } D_2 \longrightarrow D_2$	(Red IfFalse)
$D \longrightarrow D' \Rightarrow \text{if } D \text{ then } D_1 \text{ else } D_2 \longrightarrow \text{if } D' \text{ then } D_1 \text{ else } D_2$	(Red If)
$\text{null}([\]) \longrightarrow \text{true}$	(Red NullTrue)
$\text{null}(V) \longrightarrow \text{false} \quad (V \neq [\])$	(Red NullFalse)
$D \longrightarrow D' \Rightarrow \text{null}(D) \longrightarrow \text{null}(D')$	(Red Null)
$\text{labeled}(\ell \langle V \rangle, \ell) \longrightarrow \text{true}$	(Red LabeledTrue)
$\text{labeled}(V, \ell) \longrightarrow \text{false} \quad (V \neq \ell \langle V \rangle)$	(Red LabeledFalse)
$D \longrightarrow D' \Rightarrow \text{labeled}(D, \ell) \longrightarrow \text{labeled}(D', \ell)$	(Red Labeled)

Figure 7.3. Operational Semantics.

semantic function $\text{concat} : \text{Str} \times \text{Str} \rightarrow \text{Str}$ concatenates two strings. Notice that the evaluation of every construct in Figure 7.3 is fully-deterministic because proper subexpressions are always evaluated from left to right.

$D, D' \in \text{DExp} ::= \dots$	δ -Calculus + λ -Calculus
true	(True)
false	(False)
$D \oplus D'$	(String Concat)
$\text{fix } x.D$	(Fix-point)
if D_1 then D_2 else D_3	(IfThenElse)
$\text{null}(D)$	(Null Predicate)
$\text{labeled}(D, \ell)$	(Labeled Predicate)
\dots	Other primitives according to need

In the examples that follow, we use a few additional constructs that have not been introduced yet. Instead of extending the basic calculus even further, we use the primitives just defined to encode these constructs, as this does not affect the introduction of types in the next chapter. The encodings of all these constructs are standard.

$$\begin{aligned}
\text{let } x = D \text{ in } D' &\triangleq (\lambda x.D') D \\
\text{letrec } x = D \text{ in } D' &\triangleq \text{let } x = \text{fix } x.D \text{ in } D' \\
\text{not}(D) &\triangleq \text{if } D \text{ then false else true} \\
\text{and}(D, D') &\triangleq \text{if } D \text{ then (if } D' \text{ then true else false) else false} \\
\text{or}(D, D') &\triangleq \text{if } D \text{ then true else (if } D' \text{ then true else false)}
\end{aligned}$$

EXAMPLE 7.2.2. Let D_1 be defined as in Example 7.1.1. The following document expression transforms D_1 into an HTML document.

$$\begin{aligned}
&\text{letrec } G = (\lambda z.\text{if null}(z) \text{ then '' else } (z.\text{hd}/\text{receiver}) \oplus G(z.\text{tl})) \\
&\text{in} \\
&\quad \text{let } F = (\lambda x.\text{let } y = x/\text{email} \text{ in} \\
&\quad\quad \text{html}\langle \text{head}\langle \text{h3}\langle y.3/\text{subject}\rangle\rangle, \\
&\quad\quad\quad \text{body}\langle [\text{p}\langle y.1/\text{sender}\rangle, \text{p}\langle G(y.2)\rangle, \text{p}\langle y.4/\text{message}\rangle]\rangle\rangle) \\
&\quad \text{in} \\
&\quad\quad F D_1
\end{aligned}$$

Procedure G returns the string that results from concatenating the e-mail addresses of all the receivers. Procedure F returns a document whose *content* is identical to that of the input document, but whose *markup* is HTML. \square

EXAMPLE 7.2.3. In Example 7.1.1, D_1 is defined in terms of D_2 and D_3 . Let D'_1 be the result of taking D_1 and replacing D_3 by D'_3 in its definition, where

$$D'_3 = [\text{receiver}\langle \text{'santiago@cs.bu.edu'}\rangle, \text{newsgroup}\langle \text{'bu.announce'}\rangle].$$

The following document expression transforms D'_1 into an HTML document. Notice the use of the predicate labeled to extract the e-mail addresses from D'_3 .

$$\begin{aligned}
&\text{letrec } G = (\lambda z.\text{if null}(z) \text{ then ''} \\
&\quad\quad \text{else if labeled}(z.\text{hd}, \text{receiver}) \text{ then } (z.\text{hd}/\text{receiver}) \oplus G(z.\text{tl}) \\
&\quad\quad \text{else then } (z.\text{hd}/\text{newsgroup}) \oplus G(z.\text{tl})) \\
&\text{in} \\
&\quad \text{let } F = (\lambda x.\text{let } y = x/\text{email} \text{ in} \\
&\quad\quad \text{html}\langle \text{head}\langle \text{h3}\langle y.3/\text{subject}\rangle\rangle,
\end{aligned}$$

in FD'_1 `body<[p<y.1/sender>, p<G(y.2)>, p<y.4/message>]>>`

As in Example 7.2.2, procedure F returns an HTML document which is a displayable version of its input. □

Chapter 8

Typed $\delta\lambda$ -Calculus

In this chapter we introduce a typed version of the $\delta\lambda$ -Calculus presented in the last chapter. First, we present a type system without union types and show which of the examples we discussed are typable and which are not. Then, motivated by the need to type all the examples, we extend the type system (as well as the calculus) in order to support union types.

8.1 Typed $\delta\lambda$ -Calculus

The syntax of the typed $\delta\lambda$ -Calculus is obtained by simply annotating the bound variables of the constructs described in Chapter 7. Only two of those constructs bind variables, namely, abstractions $(\lambda x : \tau. D)$ and recursive definitions $(\text{fix } x : \tau. D)$. The meta-variable τ ranges over a set of types defined in Section 8.2. Additionally, we define encodings for typed versions of `let` and `letrec` as shown next.

$$\begin{aligned} \text{let } x : \tau = D \text{ in } D' &\triangleq (\lambda x : \tau. D') D \\ \text{letrec } x : \tau = D \text{ in } D' &\triangleq \text{let } x : \tau = \text{fix } x : \tau. D \text{ in } D' \end{aligned}$$

The operational semantics of the calculus is not affected by the type annotations; the reader is referred to the Figures 7.1, 7.2 and 7.3 for further details.

8.2 Type System

As in Part I, we let σ and τ range over a set of types. The syntax of types is defined inductively by the following grammar:

$\sigma, \tau \in \text{DExpType}$	$::=$	bool	(Boolean Type)
		string	(String Type)
		$\ell < \tau >$	(Label Type)
		τ^*	(List Type)
		τ_1, \dots, τ_n	(Tuple Type)
		$\tau \rightarrow \sigma$	(Arrow Type)

We assume that the type operator “*” has higher precedence than “,”, which in turn has higher precedence than “ \rightarrow ”. For example, the type $\tau^*, \sigma \rightarrow \sigma'$ is equivalent to $((\tau^*), \sigma) \rightarrow \sigma'$.

The typing rules for the $\delta\lambda$ -Calculus are presented in Figure 8.1. These rules are somewhat standard: there are rules for list introduction, (DExp Nil) and (DExp Cons), list elimination, (DExp Head) and (DExp Tail), tuple introduction (DExp Tuple) and tuple elimination (DExp Proj), as well as for label introduction (DExp Label) and label elimination (DExp Sel). The remainder of the rules correspond to the simply typed λ -Calculus, (DExp Var), (DExp Abs) and (DExp App), and the extensions proposed in Section 7.1.

EXAMPLE 8.2.1. Let D_1 be defined as in Example 7.1.1. It is immediate to prove that $\emptyset \vdash D_1 : \tau_1$ where

$$\begin{aligned} \tau_1 &= \text{email} < \tau_2 > \\ \tau_2 &= \text{sender} < \text{string} >, \text{receiver} < \text{string} >^*, \text{subject} < \text{string} >, \text{message} < \text{string} > \end{aligned}$$

Environments		
$\frac{}{\emptyset \vdash \diamond}$	$\frac{}{(\text{DEnv } \emptyset)}$	$\frac{E \vdash \diamond \quad x \notin \text{dom}(E)}{E, x : \tau \vdash \diamond}$
Expressions		
$\frac{E \vdash \diamond}{E \vdash s : \text{string}}$	$\frac{E \vdash \diamond}{E \vdash [] : \tau^*}$	$\frac{E \vdash D : \tau \quad E \vdash D' : \tau^*}{E \vdash D :: D' : \tau^*}$
$\frac{E \vdash D : \tau^*}{E \vdash D.\text{hd} : \tau}$	$\frac{E \vdash D : \tau^*}{E \vdash D.\text{tl} : \tau^*}$	$\frac{E \vdash D : \tau}{E \vdash \ell < D > : \ell < \tau >}$
$\frac{E \vdash D : \ell < \tau >}{E \vdash D / \ell : \tau}$	$\frac{E \vdash D_i : \tau_i \quad \forall i \in 1..n}{E \vdash D_1, \dots, D_n : \tau_1, \dots, \tau_n}$	$\frac{E \vdash D : \tau_1, \dots, \tau_n \quad k \in 1..n}{E \vdash D.k : \tau_k}$
$\frac{E, x : \tau, E' \vdash \diamond}{E, x : \tau, E' \vdash x : \tau}$	$\frac{E, x : \tau \vdash D : \sigma}{E \vdash \lambda x : \tau. D : \tau \rightarrow \sigma}$	$\frac{E \vdash D : \tau \rightarrow \sigma \quad E \vdash D' : \tau}{E \vdash D D' : \sigma}$
$\frac{E \vdash \diamond \quad T \in \{\text{true}, \text{false}\}}{E \vdash T : \text{bool}}$	$\frac{E \vdash D : \text{string} \quad E \vdash D' : \text{string}}{E \vdash D \oplus D' : \text{string}}$	$\frac{E, x : \tau \vdash D : \tau}{E \vdash \text{fix } x : \tau. D : \tau}$
$\frac{E \vdash D_1 : \text{bool} \quad E \vdash D_2 : \tau \quad E \vdash D_3 : \tau}{E \vdash \text{if } D_1 \text{ then } D_2 \text{ else } D_3 : \tau}$	$\frac{E \vdash D : \tau^*}{E \vdash \text{null}(D) : \text{bool}}$	$\frac{E \vdash D : \tau}{E \vdash \text{labeled}(D, \ell) : \text{bool}}$

Figure 8.1. Typing Rules for the $\delta\lambda$ -Calculus.

Let us now show that the expression $((D_1/\text{email}).2).\text{hd}/\text{receiver}$ is typable using the rules in Figure 8.1.

$\emptyset \vdash (((D_1/\text{email}).2).\text{hd})/\text{receiver} : \text{string}$	(DExp Sel)
$\emptyset \vdash ((D_1/\text{email}).2).\text{hd} : \text{receiver}\langle \text{string} \rangle$	(DExp Head)
$\emptyset \vdash (D_1/\text{email}).2 : \text{receiver}\langle \text{string} \rangle^*$	(DExp Proj)
$\emptyset \vdash D_1/\text{email} : \tau_2$	(DExp Sel)
$\emptyset \vdash D_1 : \tau_1$	(DExp Label)

Notice that the rule (DExp Sel) forces the label in the type to be identical to the label in the term, so if we replace the label `email` for `e-mail` or the label `receiver` for `recipient` in the expression shown above, this derivation would no longer hold. \square

EXAMPLE 8.2.2. In Example 7.2.2, we defined procedures G and F to transform e-mail messages into HTML. Once again, let D_1 be as defined in Example 7.1.1. Procedure G expects a list of receivers and returns a string, so its type must be of the form $\sigma_1 \rightarrow \sigma_2$ where $\sigma_1 = \text{receiver}\langle \text{string} \rangle^*$ and $\sigma_2 = \text{string}$. Let τ_1 and τ_2 be as defined in Example 8.2.1. Procedure F expects a document of type τ_1 and returns a document of type σ_3 where

$$\sigma_3 = \text{html}\langle \text{head}\langle \text{h3}\langle \text{string} \rangle \rangle, \text{body}\langle \text{p}\langle \text{string} \rangle^* \rangle \rangle$$

Using the typed encodings for `let` and `letrec`, together with the typing rules from Figure 8.1, we can prove the type of the expression shown below to be σ_3 .

```

letrec G :  $\sigma_1 \rightarrow \sigma_2 = (\lambda z : \sigma_1. \text{if null}(z) \text{ then } '' \text{ else } (z.\text{hd}/\text{receiver}) \oplus G(z.\text{tl}))$ 
in
  let F :  $\tau_1 \rightarrow \sigma_3 = (\lambda x : \tau_1. \text{let } y : \tau_2 = x/\text{email} \text{ in}$ 
    html<head<h3<y.3/subject>>,
      body<[p<y.1/sender>, p<G(y.2)>, p<y.4/message>]>>)
  in
    F D1

```

\square

Unfortunately, Example 7.2.3 from the last chapter is not typable in this system, because it is impossible to assign a type to a heterogeneous list such as `[receiver<'santiago@cs.bu.edu'>`

, newsgroup<'bu.announce'>]. Recall that this list is sugar for

$$\text{receiver}<\text{'santiago@cs.bu.edu'}>::(\text{newsgroup}<\text{'bu.announce'}>::[])$$

and that for this expression to be typable (cf. (DExp Cons)) we would need $\text{receiver}<\text{string}> = \text{newsgroup}<\text{string}>$, which is clearly impossible. A related problem lies in the definition of procedure G in the same example, specifically, in the use of the predicate `labeled`.

$$G = (\lambda z. \text{if null}(z) \text{ then } '' \\ \quad \text{else if labeled}(z.\text{hd}, \text{receiver}) \text{ then } (z.\text{hd}/\text{receiver}) \oplus G(z.\text{tl}) \\ \quad \text{else then } (z.\text{hd}/\text{newsgroup}) \oplus G(z.\text{tl}))$$

Predicate `labeled` is used to select the contents of a subtree whose label is either `receiver` or `newsgroup`. However, assuming the type of the expression $z.\text{hd}$ is either $\text{receiver}<\text{string}>$ or $\text{newsgroup}<\text{string}>$, the use of a conditional expression and the predicate `labeled` does not change the type of this expression, so $(z.\text{hd}/\text{receiver})$ and $(z.\text{hd}/\text{newsgroup})$ are still not typable. In fact, predicate `labeled` is not very useful in a typed calculus as there is no way to guarantee its proper employment without, it seems, resorting to techniques based on *flow analysis*.

In the next section we extend our type system with union types, and propose solutions to all these problems.

8.3 Type System with Union Types

In order to type some interesting examples, such as those involving the manipulation of heterogeneous lists (cf. Example 7.2.3), we need to devise a more powerful type system. In this section we present a type system with *union* types that extends the type system from the previous section.

The set of types DExpTypeU , a superset of DExpType , is defined inductively by the following

$D \longrightarrow D' \Rightarrow \text{inj}_\tau\{D\}^v \longrightarrow \text{inj}_\tau\{D'\}^v$	(Red Inj)
$\text{case inj}_{\tau_j}\{V\}^v \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow D_j\{y := V\}$ ($j \in 1..n$)	(Red CasInj)
$D \longrightarrow D' \Rightarrow$ $\text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow \text{case } D' \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n$	(Red Case)

Figure 8.2. Operational Semantics.

grammar:

$\sigma, \tau \in \text{DExpTypeU} ::=$	bool	(Boolean Type)
	string	(String Type)
	$\ell < \tau >$	(Label Type)
	τ^*	(List Type)
	τ_1, \dots, τ_n	(Tuple Type)
	$\tau \rightarrow \sigma$	(Arrow Type)
	$\tau_1 \mid \dots \mid \tau_n$	(Union Type)

We reserve the meta-variable v to range of over union types. We assume that the type operator “|” has lower precedence than “,” but higher precedence than “ \rightarrow ”. For example, the type $\tau^*, \sigma \mid \tau' \rightarrow \sigma'$ is equivalent to $((\tau^*), \sigma) \mid \tau' \rightarrow \sigma'$.

Extending the syntax of types is not sufficient, we need additional constructs at the expression level in order to work with union types. Specifically, we need constructs to *introduce* and to *eliminate* union types; these are often called *injections* and *cases*, respectively. The set DExp defined in Chapter 7 is extended as follows:

$D, D' \in \text{DExp} ::=$...	δ -Calculus + λ -Calculus + primitives
	$\text{inj}_\tau\{D\}^v$	(Injection)
	$\text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n$	(Case)

The set of values DVal is also extended to include fully-evaluated injections $\text{inj}_\tau\{V\}^v$. The reduction rules for injections and cases are shown in Figure 8.2.

We use types instead of tags as the way to indicate “where” a value is to be injected. As

$$\begin{array}{c}
\text{(DExp Inj)} \\
\frac{E \vdash D : \tau_j \quad j \in 1..n}{E \vdash \text{inj}_{\tau_j}\{D\}^v : v} \quad (v = \tau_1 \mid \dots \mid \tau_n) \\
\\
\text{(DExp Case)} \\
\frac{E \vdash D : v \quad E, y : \tau_j \vdash D_j : \sigma \quad \forall j \in 1..n}{E \vdash \text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n : \sigma} \quad (v = \tau_1 \mid \dots \mid \tau_n)
\end{array}$$

Figure 8.3. Typing Rules.

a result, the rule (Red CaseInj) relies on type equality in order to dynamically dispatch the appropriate expression during reduction of a case.¹ A smart compiler can always optimize this dispatching mechanism by creating unique tags for every component of a union type, therefore freeing the user from having to introduce artificial tags when programming with union types.

The typing rules for injections and cases are shown in Figure 8.3. The typing rule (DExp Inj) states that in order to type a term of the form $(\text{inj}_{\tau_j}\{D\}^v)$ we must prove that D has type τ_j and that τ_j is one of the types in v . The typing rule (DExp Case) requires D to have type $\tau_1 \mid \dots \mid \tau_n$ and every D_j to have type σ under the assumption that y , the variable bound by the construct, has type τ_j .

We are now in position to revisit Example 7.2.3. In the presence of union types, we can define heterogeneous lists whose types are, in general, of the form $(\tau_1 \mid \dots \mid \tau_n)^*$.

EXAMPLE 8.3.1. The types σ_i for $i \in 1..3$ and the types τ_j for $j \in 1..2$ are defined as follows:

¹Since we have not demanded a union type to be disjoint, it is possible for more than one type to match during the evaluation of a case. Should this situation arise, the evaluation of the term in question would be non-deterministic.

```

 $\sigma_1$  = receiver<string>
 $\sigma_2$  = newsgroup<string>
 $\sigma_3$  = html<head<h3<string>>, body<p<string>*>>
 $\tau_1$  = email< $\tau_2$ >
 $\tau_2$  = sender<string>, ( $\sigma_1 \mid \sigma_2$ )*, subject<string>, message<string>

```

Document expression D'_1 is identical to that of Example 7.2.3 with the exception of subexpression D'_3 , which is now defined as follows:

$$D'_3 = [\text{inj}_{\sigma_1}\{\text{receiver}\langle\text{'santiago@cs.bu.edu'}\rangle\}^{\sigma_1 \mid \sigma_2}, \text{inj}_{\sigma_2}\{\text{newsgroup}\langle\text{'bu.announce'}\rangle\}^{\sigma_1 \mid \sigma_2}]$$

The terms `receiver<'santiago@cs.bu.edu'>` and `newsgroup<'bu.announce'>` are injected using the types σ_1 and σ_2 , respectively. Clearly, the type of D'_3 is $(\sigma_1 \mid \sigma_2)^*$. Let us now rewrite procedures F and G from Example 7.2.3 using union types (in fact, only G needs to be re-written).

```

letrec G : ( $\sigma_1 \mid \sigma_2$ )*  $\rightarrow$  string =
  ( $\lambda z : (\sigma_1 \mid \sigma_2)^*$ .if null(z) then ''
    else if case z.hd bind y  $\sigma_1 \Rightarrow$  (y/receiver)  $\oplus$  G(z.tl)
               $\sigma_2 \Rightarrow$  (y/newsgroup)  $\oplus$  G(z.tl))
in
  let F :  $\tau_1 \rightarrow \sigma_3$  = ( $\lambda x : \tau_1$ .let y :  $\tau_2 = x/\text{email}$  in
    html<head<h3<y.3/subject>>,
      body<[p<y.1/sender>, p<G(y.2)>, p<y.4/message>]>>
  in
    F D'_1

```

Notice the use of the construct `case` in order to check the type of a list element before selecting its contents. □

8.4 Subject Reduction

In this section we prove a Subject Reduction result for the typed $\delta\lambda$ -Calculus. The Subject Reduction theorem states that if an expression is well-typed, then it will only reduce to another well-typed expression. In particular, a reduct will never be a meaningless expression allowed by the syntax, e.g., an expression like $(\ell_1 \langle D_1 \rangle, \ell_2 \langle D_2 \rangle) / \ell_1$ or $(\lambda x : \tau. D) \cdot \text{hd}$.

Lemma 8.4.1 (Substitution). *Let D and D' be document expressions. If $E, x : \sigma, E' \vdash D : \tau$ and $E, E' \vdash D' : \sigma$ then $E, E' \vdash D\{x := D'\} : \tau$.* □

Proof. By induction on derivations. □

Theorem 8.4.2 (Subject Reduction). *Let D and D' be document expressions. If $D \longrightarrow D'$ and $E \vdash D : \tau$ then $E \vdash D' : \tau$.* □

Proof. By induction on derivations using the rules from Figures 7.1, 7.2, 7.3 and 8.2.

(Red Head) By assumption we have $(V :: V') \cdot \text{hd} \longrightarrow V$ and $E \vdash (V :: V') \cdot \text{hd} : \tau$. Then, it follows by (DExp Head) that $E \vdash (V :: V') : \tau*$, and by (DExp Cons) that $E \vdash V : \tau$ as desired.

(Red Tail) By assumption we have $(V :: V') \cdot \text{tl} \longrightarrow V'$ and $E \vdash (V :: V') \cdot \text{tl} : \tau$. Then, it follows by (DExp Tail) that $E \vdash (V :: V') : \tau$, and by (DExp Cons) that $E \vdash V' : \tau$ as desired.

(Red Proj) By assumption we have $(V_1, \dots, V_n) \cdot k \longrightarrow V_k$ with $k \in 1..n$, and $E \vdash (V_1, \dots, V_n) \cdot k : \tau_k$. Then, it follows by (DExp Proj) that $E \vdash (V_1, \dots, V_n) : \tau_1, \dots, \tau_n$, and by (DExp Tuple) that $E \vdash V_k : \tau_k$ as desired.

(Red Sel) By assumption we have $(\ell \langle V \rangle) / \ell \longrightarrow V$ and $E \vdash (\ell \langle V \rangle) / \ell : \tau$. Then, it follows by (DExp Sel) that $E \vdash \ell \langle V \rangle : \ell \langle \tau \rangle$, and by (DExp Label) that $E \vdash V : \tau$ as desired.

(Red DHead) By assumption we have $D.\text{hd} \longrightarrow D'.\text{hd}$ because $D \longrightarrow D'$ and $E \vdash D.\text{hd} : \tau$.

Then, it follows by (DExp Head) that $E \vdash D : \tau^*$. By induction hypothesis we have $E \vdash D' : \tau^*$, and by (DExp Head) once again we get $E \vdash D'.\text{hd} : \tau$ as desired.

(Red DTail) By assumption we have $D.\text{tl} \longrightarrow D'.\text{tl}$ because $D \longrightarrow D'$ and $E \vdash D.\text{tl} : \tau$. Then,

it follows by (DExp Tail) that $E \vdash D : \tau$. By induction hypothesis we have $E \vdash D' : \tau$, and by (DExp Tail) once again we get $E \vdash D'.\text{tl} : \tau$ as desired.

(Red DProj) By assumption we have $D.k \longrightarrow D'.k$ because $D \longrightarrow D'$ and $E \vdash D.k : \tau$.

Then, it follows by (DExp Proj) that $E \vdash D : \tau_1, \dots, \tau_n$ with $k \in 1..n$ and $\tau = \tau_k$. By induction hypothesis we have $E \vdash D' : \tau_1, \dots, \tau_n$, and by (DExp Proj) once again we get $E \vdash D'.k : \tau$ as desired.

(Red DSel) By assumption we have $D/\ell \longrightarrow D'/\ell$ because $D \longrightarrow D'$ and $E \vdash D/\ell : \tau$.

Then, it follows by (DExp Sel) that $E \vdash D : \ell < \tau >$. By induction hypothesis we have $E \vdash D' : \ell < \tau >$, and by (DExp Sel) once again we get $E \vdash D'/\ell : \tau$ as desired.

(Red ConsHead) By assumption we have $D :: D'' \longrightarrow D' :: D''$ because $D \longrightarrow D'$ and $E \vdash$

$D :: D'' : \tau$. Then, it follows by (DExp Cons) that $E \vdash D : \sigma$ with $\tau = \sigma^*$ and that $E \vdash D'' : \tau$. By induction hypothesis we have $E \vdash D' : \sigma$, and by (DExp Cons) once again we get $E \vdash D' :: D'' : \tau$ as desired.

(Red ConsTail) By assumption we have $V :: D \longrightarrow V :: D'$ because $D \longrightarrow D'$ and $E \vdash V :: D : \tau$.

Then, it follows by (DExp Cons) that $E \vdash V : \sigma$ with $\tau = \sigma^*$ and that $E \vdash D : \tau$. By induction hypothesis we have $E \vdash D' : \tau$, and by (DExp Cons) once again we get $E \vdash V :: D' : \tau$ as desired.

(Red Label) By assumption we have $\ell\langle D \rangle \longrightarrow \ell\langle D' \rangle$ because $D \longrightarrow D'$ and $E \vdash \ell\langle D \rangle : \tau$.

Then, it follows by (DExp Label) that $E \vdash D : \sigma$ with $\tau = \ell\langle \sigma \rangle$. By induction hypothesis we have $E \vdash D' : \sigma$, and by (DExp Label) once again we get $E \vdash \ell\langle D' \rangle : \tau$ as desired.

(Red Tuple) By assumption we have $V_1, \dots, V_{k-1}, D_k, \dots, D_n \longrightarrow V_1, \dots, V_{k-1}, D'_k, \dots, D_n$ because $D_k \longrightarrow D'_k$ and $E \vdash V_1, \dots, V_{k-1}, D_k, \dots, D_n : \tau$. Then, it follows by (DExp Tuple) that $E \vdash D_k : \tau_k$ with $\tau = \tau_1, \dots, \tau_n$ and $k \in 1..n$. By induction hypothesis we have $E \vdash D'_k : \tau_k$, and by (DExp Tuple) once again we get $E \vdash V_1, \dots, V_{k-1}, D'_k, \dots, D_n : \tau$ as desired.

(Red Beta) By assumption we have $(\lambda x : \sigma. D)V \longrightarrow D\{x := V\}$ and $E \vdash (\lambda x : \sigma. D)V : \tau$. Then, it follows by (DExp App) that $E \vdash (\lambda x : \sigma. D) : \sigma \rightarrow \tau$ and $E \vdash V : \sigma$. Hence, by (DExp Abs) we get $E, x : \sigma \vdash D : \tau$ and, from the last two judgements using Lemma 8.4.1, we have $E \vdash D\{x := V\} : \tau$ as desired.

(Red AppLeft) By assumption we have $DD'' \longrightarrow D'D''$ because $D \longrightarrow D'$ and $E \vdash DD'' : \tau$. Then, it follows by (DExp App) that $E \vdash D : \sigma \rightarrow \tau$ and $E \vdash D'' : \sigma$. By induction hypothesis we have $E \vdash D' : \sigma \rightarrow \tau$, and by (DExp App) once again we get $E \vdash D'D'' : \tau$ as desired.

(Red AppRight) Similar to (Red AppLeft).

(Red Fix) By assumption we have $\text{fix } x : \tau. D \longrightarrow D\{x := \text{fix } x : \tau. D\}$ and $E \vdash \text{fix } x : \tau. D : \tau$. Then, it follows by (DExp Fix) that $E, x : \tau \vdash D : \tau$. Hence, from the last two judgements using Lemma 8.4.1, we have $E \vdash D\{x := \text{fix } x : \tau. D\} : \tau$ as desired.

(Red Concat) Immediate from the fact that concat has type $\text{Str} \times \text{Str} \rightarrow \text{Str}$, and that by (DExp Str)

every element in Str has type string.

(Red ConcatLeft) By assumption we have $D \oplus D'' \longrightarrow D' \oplus D''$ because $D \longrightarrow D'$ and $E \vdash D \oplus D'' : \tau$. Then, it follows that $\tau = \text{string}$ and that by (DExp Concat) we have $E \vdash D : \tau$ and $E \vdash D'' : \tau$. By induction hypothesis we have $E \vdash D' : \tau$, and by (DExp Concat) once again we get $E \vdash D' \oplus D'' : \tau$ as desired.

(Red ConcatRight) Similar to (Red ConcatLeft).

(Red IfTrue) By assumption we have if true then D_1 else D_2 and $E \vdash \text{if true then } D_1 \text{ else } D_2 : \tau$. Then, it follows by (DExp If) that $E \vdash D_1 : \tau$ as desired.

(Red IfFalse) Similar to (Red IfTrue).

(Red If) By assumption we have if D then D_1 else $D_2 \longrightarrow \text{if } D' \text{ then } D_1 \text{ else } D_2$ because $D \longrightarrow D'$ and $E \vdash \text{if } D \text{ then } D_1 \text{ else } D_2 : \tau$. Then, it follows by (DExp If) we have $E \vdash D : \text{bool}$ and $E \vdash D_i : \tau$ for $i \in 1..2$. By induction hypothesis we have $E \vdash D' : \text{bool}$, and by (DExp If) once again we get $E \vdash \text{if } D' \text{ then } D_1 \text{ else } D_2 : \tau$ as desired.

(Red NullTrue) Immediate using rules (DExp Null) and (DExp Bool).

(Red NullFalse) Similar to (Red NullTrue).

(Red Null) By assumption we have $\text{null}(D) \longrightarrow \text{null}(D')$ because $D \longrightarrow D'$ and $E \vdash \text{null}(D) : \tau$. Then, it follows by (DExp Null) that $E \vdash D : \sigma^*$. By induction hypothesis we have $E \vdash D' : \sigma^*$, and by (DExp Null) once again we get $E \vdash \text{null}(D') : \tau$ as desired.

(Red LabeledTrue) Immediate using rules (DExp Labeled) and (DExp Bool).

(Red LabeledFalse) Similar to (Red LabeledTrue).

(Red Labeled) By assumption we have $\text{labeled}(D, \ell) \longrightarrow \text{labeled}(D', \ell)$ because $D \longrightarrow D'$ and $E \vdash \text{labeled}(D, \ell) : \tau$. Then, it follows by (DExp Labeled) that $E \vdash D : \sigma$. By induction hypothesis we have $E \vdash D' : \sigma$, and by (DExp Labeled) once again we get $E \vdash \text{labeled}(D', \ell) : \tau$ as desired.

(Red Inj) By assumption we have $\text{inj}_{\tau_j}\{D\}^v \longrightarrow \text{inj}_{\tau_j}\{D'\}^v$ because $D \longrightarrow D'$ and $E \vdash \text{inj}_{\tau_j}\{D\}^v : \tau$. Then, it follows that $\tau = v = \tau_1 \mid \dots \mid \tau_n$ with $j \in 1..n$, and by (DExp Inj) that $E \vdash D : \tau_j$. By induction hypothesis we have $E \vdash D' : \tau_j$, and by (DExp Inj) once again we get $E \vdash \text{inj}_{\tau_j}\{D'\}^v : \tau$ as desired.

(Red CaseInj) By assumption we have $\text{case inj}_{\tau_j}\{V\}^v \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow D_j\{y := V\}$ with $j \in 1..n$, and $E \vdash \text{case inj}_{\tau_j}\{V\}^v \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n : \tau$. Then, it follows by (DExp Case) that $E \vdash \text{inj}_{\tau_j}\{V\}^v : v$ with $v = \tau_1 \mid \dots \mid \tau_n$ and that $E, y : \tau_i \vdash D_i : \tau$ for every $i \in 1..n$. Hence, from (DExp Inj) we derive $E \vdash V : \tau_j$ and, from the last two judgements using Lemma 8.4.1, we get $E \vdash D_j\{y := V\} : \tau$ as desired.

(Red Case) By assumption we have $\text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow \text{case } D' \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n$ because $D \longrightarrow D'$ and $E \vdash \text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n : \tau$. Then, it follows that by (DExp Case) that $E \vdash D : v$ with $v = \tau_1 \mid \dots \mid \tau_n$ and that $E, y : \tau_i \vdash D_i : \tau$ for every $i \in 1..n$. By induction hypothesis we have $E \vdash D' : v$, and by (DExp Case) once again we get $E \vdash \text{case } D' \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n : \tau$ as desired. \square

The Subject Reduction theorem asserts that *if an expression D is typable and it reduces to an expression D' , then the latter is typable too*. A stronger result, sometimes referred to as Absence of Stuck States theorem, shows that *if an expression D that is not a value is typable, then it reduces to an expression D' and this reduct is typable too*. A consequence of this theorem is that

every typable expression that does not diverge reduces to a value.

Unfortunately, the Absence of Stuck States theorem does not hold for the typed $\delta\lambda$ -Calculus: for example, the term $([\] \cdot \text{hd})$ is typable, does not diverge and cannot be reduced any further even though it is not a value.

Chapter 9

From XML to the $\delta\lambda$ -Calculus

In Section 7.1, we explained the differences between the syntax of an XML document and the syntax of a $\delta\lambda$ -Calculus expression. These syntactic differences are not superficial, because in the $\delta\lambda$ -Calculus we require the programmer to provide additional information to hint at the intended usage of a document (e.g. we require the programmer to insert square brackets to delimit unbounded sequences). Although this additional information is not present in a bona fide XML document, it is possible to recover it by analyzing its associated DTD —which in the discussion that follows we assume is always available.

In this chapter we present an algorithm that takes an XML document and a type, and returns the $\delta\lambda$ -Calculus expression that results from inserting all the annotations as dictated by the type. These annotations are of two different kinds: (i) square brackets that are needed to define unbounded sequences and (ii) injections that are needed to introduce union types. The existence of this algorithm has interesting practical consequences, for example, it ensures that external XML documents can be imported and manipulated in a statically-typed framework.

We conclude this chapter by presenting another algorithm that works in the opposite direc-

tion as the one described above. Namely, it takes a $\delta\lambda$ -Calculus expression (more precisely a value) and returns an XML document; this algorithm can be used to export documents out of the $\delta\lambda$ -Calculus framework.

9.1 Documents and DTD Types

Let us begin by introducing the syntax of XML documents, as well as that of their associated DTD types.

$$\begin{array}{ll}
 A, B \in \text{XDoc} & ::= \epsilon & \text{(Empty)} \\
 & | s & \text{(String)} \\
 & | \ell \langle A \rangle & \text{(Label)} \\
 & | A_1, \dots, A_n \quad (n \geq 2) & \text{(Tuple)}
 \end{array}$$

The set XDoc can be regarded as a subset of DExp in which $[]$ is replaced by ϵ . As in previous chapters, and in an attempt to keep the formalism simple, we ignore features such as attributes, namespaces, processing instructions, etc., as they are not essential for our presentation.

We assume that operator “,” is associative, so two documents are equal modulo re-grouping inside a tuple. For example, we have $(\ell \langle A_1, (A_2, A_3) \rangle, A_4), A_5 = \ell \langle A_1, A_2, A_3 \rangle, A_4, A_5$. Additionally, we identify documents in XDoc modulo a congruence relation \cong based on the axioms $A, \epsilon \cong A$ and $\epsilon, A \cong A$. For example, we have that the document $A_1, \epsilon, \epsilon, A_2, \epsilon$ is congruent to A_1, A_2 . The need for this congruence relation will become apparent when we define DTD validation.

The set XDocTypeU defined below is a proper subset of DExpTypeU defined in Section 8.3. We use the same Greek letters to range over XDocTypeU ; the context will always desambiguate a reference to an element of either set.

$\sigma, \tau \in \text{XDocTypeU} ::=$	<code>string</code>	(String Type)
	$\ell \langle \tau \rangle$	(Label Type)
	τ^*	(List Type)
	$\tau_1, \dots, \tau_n \quad (n \geq 2)$	(Tuple Type)
	$\tau_1 \mid \dots \mid \tau_n \quad (n \geq 2)$	(Union Type)

The denotation of a type in XDocTypeU is a set of XML documents. Let the denotation function $\llbracket \cdot \rrbracket : \text{XDocTypeU} \rightarrow 2^{\text{XDoc}}$ be defined as follows:

1. $\llbracket \text{string} \rrbracket = \text{Str}$,
2. $\llbracket \ell \langle \tau \rangle \rrbracket = \{\ell \langle A \rangle \mid A \in \llbracket \tau \rrbracket\}$,
3. $\llbracket \tau^* \rrbracket = \{\epsilon\} \cup (\bigcup_{n \geq 1} \{A_1, \dots, A_n \mid A_i \in \llbracket \tau \rrbracket \text{ for } i \in 1..n\})$,
4. $\llbracket \tau_1, \dots, \tau_n \rrbracket = \{A_1, \dots, A_n \mid A_i \in \llbracket \tau_i \rrbracket \text{ for } i \in 1..n\}$,
5. $\llbracket \tau_1 \mid \dots \mid \tau_n \rrbracket = \bigcup_{i=1}^n \{A \mid A \in \llbracket \tau_i \rrbracket\}$.

Intuitively, we would like to say that a document A validates a DTD type τ iff $A \in \llbracket \tau \rrbracket$. Unfortunately, this definition does not capture our intuition completely, for example, the document $\ell_1 \langle s \rangle$ does not validate the type $\ell_1 \langle \text{string} \rangle, \ell_2 \langle \text{string} \rangle^*$ because the former is a labeled document and the latter a tuple type. On the other hand, it follows from the definition of \cong that $\ell_1 \langle s \rangle \cong \ell_1 \langle s \rangle, \epsilon$ and from the denotation of a DTD type that $\ell_1 \langle s \rangle, \epsilon \in \llbracket \ell_1 \langle \text{string} \rangle, \ell_2 \langle \text{string} \rangle^* \rrbracket$. Given a document A and a type τ , we write $A \in_{\cong} \llbracket \tau \rrbracket$ iff there exists a document $A' \cong A$ such that $A' \in \llbracket \tau \rrbracket$.

In case 4 of the definition shown above, the number of A 's on the right hand side has to be the same as the number of τ 's on the left hand side. At first glance, it looks as if a document such as $s_1, s_2, \ell \langle s_3 \rangle$ where the s_i 's are in Str is not in the denotation of $\text{string}^*, \ell \langle \text{string} \rangle$ because

the former has length 3 and the latter has length 2. However, since operator “,” is associate, we have $s_1, s_2, \ell \langle s_3 \rangle = (s_1, s_2), \ell \langle s_3 \rangle$ and then that $(s_1, s_2), \ell \langle s_3 \rangle \in \llbracket \text{string}^*, \ell \langle \text{string} \rangle \rrbracket$ as expected.

Definition 9.1.1 (DTD Validation). Let A be an XML document and τ a DTD type. We say that A *validates* τ iff $A \in_{\cong} \llbracket \tau \rrbracket$. □

Even though A validating τ does not imply that $A \in \llbracket \tau \rrbracket$, the converse does hold: if $A \in \llbracket \tau \rrbracket$ then A validates τ . The last statement follows trivially from the fact that \cong is a congruence relation.

From a practical point of view, there is no need to worry about the associativity of “,” (as XML provides no means for grouping) or the appearance of an ϵ in the middle of a tuple (as ϵ 's are not part of the syntax of real XML documents). From our perspective, document ϵ is only needed to denote (i) the empty XML document and (ii) an empty element such as `<subject/>`.¹ For this reason, we define the *canonical form* of a document A as that in which every tuple A_1, \dots, A_n in A has the property that A_i is not itself a tuple and $A_i \neq \epsilon$ for $i \in 1..n$. Clearly, for every document A there exists a document A' such that $A \cong A'$ and A' is in canonical form.

9.2 Well-formed DTD Types

The description as well as the implementation of various algorithms that operate on types from XDocTypeU is greatly simplified by imposing a few additional restrictions on their structure. Our definition of a *well-formed* DTD type, to be presented shortly, is akin to that proposed by the W3C in [Wora].

¹In XML, the markup `<subject/>` is a convenient shorthand for `<subject></subject>`. In our syntax, the document `<subject/>` is written as `subject < ϵ >`.

Definition 9.2.1 (Nullable Types). A DTD type τ is nullable iff $\text{nullable}(\tau) = \text{true}$. The predicate $\text{nullable}()$ is defined by induction on the structure of DTD types:

1. $\text{nullable}(\text{string}) = \text{false}$,
2. $\text{nullable}(\ell \langle \tau \rangle) = \text{false}$,
3. $\text{nullable}(\tau^*) = \text{true}$,
4. $\text{nullable}(\tau_1, \dots, \tau_n) = \bigwedge_{i=1}^n \text{nullable}(\tau_i)$,
5. $\text{nullable}(\tau_1 \mid \dots \mid \tau_n) = \bigvee_{i=1}^n \text{nullable}(\tau_i)$. □

Next, we define the function $\text{first}()$ on XML documents and on DTD types. Technically, these are two different functions, but we use the same name because of their similarity and because it is always possible to disambiguate them.

Definition 9.2.2 (Function $\text{first}()$). We reserve the labels \flat and \sharp from the set Lab . The definition of $\text{first}()$ is by induction on the structure of its input.

On documents:

1. $\text{first}(\epsilon) = \{\flat\}$,
2. $\text{first}(s) = \{\sharp\}$,
3. $\text{first}(\ell \langle A \rangle) = \{\ell\}$,
4. $\text{first}(A_1, \dots, A_n) = \text{first}(A_1)$.

On types:

1. $\text{first}(\text{string}) = \{\sharp\}$,

2. $\text{first}(\ell \langle \tau \rangle) = \{\ell\}$,
3. $\text{first}(\tau^*) = \{b\} \cup \text{first}(\tau)$,
4. $\text{first}(\tau_1, \dots, \tau_n) = \bigcup_{j=1}^m \text{first}(\tau_j)$ where $m = \min(\{k \mid \text{nullable}(\tau_1, \dots, \tau_k) = \text{false}\} \cup \{n\})$,
5. $\text{first}(\tau_1 \mid \dots \mid \tau_n) = \bigcup_{j=1}^n \text{first}(\tau_j)$. □

Notice that function $\text{first}()$ always returns a singleton set when applied to a document. We shall return to the relationship between the two definitions of $\text{first}()$ after introducing one more definition.

The last two definitions, predicate $\text{nullable}()$ and function $\text{first}()$, are needed to formalize the conditions under which a DTD type is said to be well formed.

Definition 9.2.3 (Well-formed DTD Types). The well-formedness condition is defined by induction on the structure of DTD types:

1. string is well formed,
2. $\ell \langle \tau \rangle$ is well formed iff τ is well formed,
3. τ^* is well formed iff (i) τ is well formed and (ii) τ is not a list type,
4. τ_1, \dots, τ_n is well formed iff for $i \in 1..n$ (i) the types τ_i are well formed and (ii) τ_i is not a tuple type and (iii) for every $j, k \in 1..n$, such that $j < k$, if $\text{first}(\tau_j) \cap \text{first}(\tau_k) \neq \emptyset$ then $\text{nullable}(\tau_j, \dots, \tau_{k-1}) = \text{false}$,
5. $\tau_1 \mid \dots \mid \tau_n$ is well formed iff for $i \in 1..n$ (i) the types τ_i are well formed and (ii) τ_i is not a union type and (iii) the sets $\text{first}(\tau_i)$ are pairwise disjoint.

Condition (ii) in the last three cases simply states that the type constructors “*”, “,” and “|” cannot be immediately nested. Clearly, this syntactic requirement does not impose any semantic restrictions (e.g. in the case of “,” and “|” it follows from the associativity of these operators). \square

Despite its apparent complexity, the definition of a well-formed DTD is very intuitive.² Let us list a few examples of DTD types that are *not* well formed:

- $\sigma_1 = \ell_1 < \tau_1 > *, \ell_1 < \tau_2 >$ is not well formed since $\text{first}(\ell_1 < \tau_1 > *) \cap \text{first}(\ell_1 < \tau_2 >) = \{\ell_1\}$ and $\text{nullable}(\ell_1 < \tau_1 > *) = \text{true}$. Hence, condition 4 is violated.
- $\sigma_2 = \ell_1 < \tau_1 > *, \ell_2 < \tau_2 > *$ is not well formed since $\text{first}(\ell_1 < \tau_1 > *) \cap \text{first}(\ell_2 < \tau_2 > *) = \{b\}$ and $\text{nullable}(\ell_1 < \tau_1 > *) = \text{true}$. Hence, condition 4 is violated.
- $\sigma_3 = \text{string} | \text{string}*$ is not well formed as $\text{first}(\text{string}) \cap \text{first}(\text{string}*) = \{\#\}$. Hence, condition 5 is violated.
- $\sigma_4 = \ell_1 < \tau_1 > * | \ell_2 < \tau_2 > | \ell_1 < \tau_3 >$ is not well formed as $\text{first}(\ell_1 < \tau_1 > *) \cap \text{first}(\ell_1 < \tau_3 >) = \{\ell_1\}$. Hence, condition 5 is violated.
- Any type built from σ_i for $i \in 1..4$ is not well formed.

Notice that a type such as $\ell_1 < \tau_1 > *, \ell_2 < \tau_2 >, \ell_1 < \tau_3 > *$ is well formed because even though the set $\text{first}(\ell_1 < \tau_1 > *) \cap \text{first}(\ell_1 < \tau_3 > *)$ is non-empty we have $\text{nullable}(\ell_1 < \tau_1 > *, \ell_2 < \tau_2 >) = \text{false}$.

The following two lemmas are needed in the proofs of Section 9.4. Lemma 9.2.5, in particular, relates the two definitions of $\text{first}()$ together with the notion of DTD validation.

Lemma 9.2.4 (Property of well-formed tuple types). *If τ_1, \dots, τ_n is a well-formed DTD type then $\text{nullable}(\tau_1, \dots, \tau_n) = \text{false}$.* \square

²If we regard a DTD type as a grammar, then a well-formed DTD type would correspond to a grammar that can be parsed using a non-backtracking left-to-right parser.

Proof. From the definition of DTD types we know that $n \geq 2$. Suppose $\text{nullable}(\tau_1, \dots, \tau_n) = \text{true}$. Then, it must be $\text{nullable}(\tau_i) = \text{true}$ for every $i \in 1..n$. Hence, we have $\text{first}(\tau_1) \cap \text{first}(\tau_2) \supseteq \{\epsilon\}$ which implies that the DTD type τ_1, \dots, τ_n is not well formed. Contradiction, so it must be $\text{nullable}(\tau_1, \dots, \tau_n) = \text{false}$. \square

Lemma 9.2.5 (Property of $\text{first}()$). *Let A be an XML document in canonical form and τ a well-formed DTD type. If A validates τ then $\text{first}(A) \subseteq \text{first}(\tau)$.* \square

Proof. By induction on the structure of τ .

($\tau = \text{string}$) By definition of DTD validation we have $A \in \text{Str}$. It follows from the definition of $\text{first}()$ that $\text{first}(A) = \text{first}(\text{string}) = \{\#\}$.

($\tau = \ell \langle \tau' \rangle$) By definition of DTD validation we have $A = \ell \langle A' \rangle$ for some A' . It follows from the definition of $\text{first}()$ that $\text{first}(\ell \langle A' \rangle) = \text{first}(\ell \langle \tau' \rangle) = \{\ell\}$.

($\tau = \tau'^*$) By definition of DTD validation, and the assumption that A is in canonical form, we have that either $A = \epsilon$ or $A = A_1, \dots, A_p$ with $p \geq 1$ and $A_i \neq \epsilon$ for $i \in 1..p$. If $A = \epsilon$ then by definition of $\text{first}()$ we have $\text{first}(A) = \{\epsilon\} \subseteq \text{first}(\tau'^*)$. If $A = A_1, \dots, A_p$ then it must be $A_i \in \llbracket \tau' \rrbracket$ for $i \in 1..p$. By induction hypothesis and the definition of $\text{first}()$, we get $\text{first}(A) = \text{first}(A_1) \subseteq \text{first}(\tau') \subseteq \text{first}(\tau'^*)$.

($\tau = \tau_1, \dots, \tau_n$) By definition of DTD validation, and the assumption that A is in canonical form, we have $A = A_1, \dots, A_p$ with $p \geq 1$ and $A_i \neq \epsilon$ for $i \in 1..p$. Define $m = \min(\{k \mid \text{nullable}(\tau_1, \dots, \tau_k) = \text{false}\} \cup \{n\})$. Since A validates τ_1, \dots, τ_n there must be a $j \in 1..n$ such that $A_1 \in \llbracket \tau_j \rrbracket$ and, consequently, such that A_1 validates τ_j . By induction hypothesis we have $\text{first}(A) = \text{first}(A_1) \subseteq \text{first}(\tau_j)$. If $j \leq m$ then by definition of

first() we get $\text{first}(\tau_j) \subseteq \bigcup_{h=1}^m \text{first}(\tau_h) = \text{first}(\tau_1, \dots, \tau_n)$. Suppose $j > m$. By construction, $\text{nullable}(\tau_m) = \text{false}$ and A validates τ_1, \dots, τ_n , so A cannot be of the form A_1, \dots, A_p as $A_1 \notin \llbracket \tau_m \rrbracket$. Contradiction, that resulted from assuming $j > m$, so it must be $j \leq m$ and then $\text{first}(\tau_j) \subseteq \bigcup_{h=1}^m \text{first}(\tau_h) = \text{first}(\tau_1, \dots, \tau_n)$.

$(\tau = \tau_1 \mid \dots \mid \tau_n)$ By definition of DTD validation there exists a $k \in 1..n$ such that A validates τ_k .

By induction hypothesis and the definition of first(), we conclude that $\text{first}(A) \subseteq \text{first}(\tau_k) \subseteq \bigcup_{j=1}^n \text{first}(\tau_j) = \text{first}(\tau_1 \mid \dots \mid \tau_n)$. \square

9.3 Algorithm IMP

Algorithm IMP, see Figure 9.1, takes a well-formed DTD type and a document in canonical form and returns the $\delta\lambda$ -Calculus expression that results from inserting all the annotations as dictated by the type. Specifically, the input to Algorithm IMP is a pair of the form $(\sigma; A)$ where σ is a DTD type and A is a document, and the output is a pair of the form $(D; B)$ where D is a $\delta\lambda$ -Calculus expression and B the suffix of A not validating σ .

Algorithm IMP is defined inductively on the structure of its input DTD type. The five different cases are defined by matching the structure of the type and the structure of the document. For the latter, we use the pattern A_1, \dots, A_m with $m \geq 1$ to match any sequence of documents. For example, the document $B = s_1, \ell < s_2 >$ matches A_1, \dots, A_m with A_1 bound to s_1 , A_2 bound to $\ell < s_2 >$ and m bound to 2. The main advantage of using these patterns is the ability to group similar cases, e.g. A_1, \dots, A_m also matches ϵ with A_1 bound to ϵ and m bound to 1. Hereinafter, we assume that a tuple of the form A_i, \dots, A_j is equal to ϵ whenever $i > j$.

Algorithm IMP assumes that either a non-empty prefix of the input document or the docu-

$\text{IMP}(\text{string}; A_1, \dots, A_m)$	$=$	$(A_1; A_2, \dots, A_m)$
$\text{IMP}(\ell < \tau >; A_1, \dots, A_m)$	$=$	$\text{let } \ell < A' > = A_1$ $(D; B) = \text{IMP}(\tau; A')$ in $(\ell < D >; A_2, \dots, A_m)$
$\text{IMP}(\tau^*; A_1, \dots, A_m)$	$=$	$\text{if } A_1, \dots, A_m = \epsilon \text{ or } \text{first}(A_1) \not\subseteq \text{first}(\tau) \text{ then}$ $([]; A_1, \dots, A_m)$ else $\text{let } (D_1; B_1) = \text{IMP}(\tau; A_1, \dots, A_m)$ $(D_2; B_2) = \text{IMP}(\tau^*; B_1)$ in $(D_1 :: D_2; B_2)$ fi
$\text{IMP}(\tau_1, \dots, \tau_n; A_1, \dots, A_m)$	$=$	$\text{if } \text{first}(A_1) \not\subseteq \text{first}(\tau_1) \text{ then}$ $\text{let } (D_1; B_1) = \text{IMP}(\tau_1; \epsilon)$ $(D_2; B_2) = \text{IMP}(\tau_2, \dots, \tau_n; A_1, \dots, A_m)$ in $(D_1, D_2; B_2)$ else $\text{let } (D_1; B_1) = \text{IMP}(\tau_1; A_1, \dots, A_m)$ $(D_2; B_2) = \text{IMP}(\tau_2, \dots, \tau_n; B_1)$ in $(D_1, D_2; B_2)$ fi
$\text{IMP}(\tau_1 \mid \dots \mid \tau_n; A_1, \dots, A_m)$	$=$	$\text{let } j \in \{h \in 1..n \mid \text{first}(A_1) \subseteq \text{first}(\tau_h)\}$ $(D; B) = \text{IMP}(\tau_j; A_1, \dots, A_m)$ $v = \tau_1 \mid \dots \mid \tau_n$ in $(\text{inj}_{\tau_j} \{D\}^v; B)$

Figure 9.1. Algorithm IMP.

ment ϵ validate the input type. The cases in which the input DTD type is either string or $\ell < \tau >$ are immediate: their correctness follows from the fact that the types string and $\ell < \tau >$ are non-nullable (cf. Section 9.4).

The case for τ^* is divided into one for τ and another for τ^* if the prefix of A_1, \dots, A_m that validates τ^* is non-empty (which can only happen if $A_1, \dots, A_m \neq \epsilon$). For the other two cases, i.e. either $A_1, \dots, A_m = \epsilon$ or an empty prefix of A_1, \dots, A_m validating τ^* , the expression $[]$ is

returned without advancing the input.

The case for τ_1, \dots, τ_n is divided into one for τ_1 and another for τ_2, \dots, τ_n . Should there be no input validating τ_1 , a recursive call on ϵ is inserted before analyzing the case for τ_2, \dots, τ_n .

At last, the case for $\tau_1 \mid \dots \mid \tau_n$ determines which part of the union is validated by the input document and proceeds recursively. Notice that the set $\{k \in 1..n \mid \text{first}(A_1) \subseteq \text{first}(\tau_k)\}$ is always a singleton because by assumption the input type is well formed.

EXAMPLE 9.3.1. Let $\sigma = \ell_1 \langle \text{string} \rangle, (\text{string}^* \mid \ell_2 \langle \text{string} \rangle), \ell_3 \langle \text{string} \rangle^*$ and $A = \ell_1 \langle s_1 \rangle, \ell_3 \langle s_3 \rangle$. Clearly, σ is well formed and A is in canonical form. Moreover, since $A \cong \ell_1 \langle s_1 \rangle, \epsilon, \ell_3 \langle s_3 \rangle$ we have that A validates σ . Using the rules in Figure 9.1 we get:

$$\begin{aligned}
\text{IMP}(\sigma; A) &= (\ell_1 \langle s_1 \rangle, \text{inj}_{\text{string}^*} \{[\]\}^v, [\ell_3 \langle s_3 \rangle]; \epsilon) \\
&\hookrightarrow \text{IMP}(\ell_1 \langle \text{string} \rangle; \ell_1 \langle s_1 \rangle, \ell_3 \langle s_3 \rangle) = (\ell_1 \langle s_1 \rangle; \ell_3 \langle s_3 \rangle) \\
&\quad \hookrightarrow \text{IMP}(\text{string}; s_1) = (s_1; \epsilon) \\
&\hookrightarrow \text{IMP}((\text{string}^* \mid \ell_2 \langle \text{string} \rangle), \ell_3 \langle \text{string} \rangle^*; \ell_3 \langle s_3 \rangle) = (\text{inj}_{\text{string}^*} \{[\]\}^v, [\ell_3 \langle s_3 \rangle]; \epsilon) \\
&\quad \hookrightarrow \text{IMP}(\text{string}^* \mid \ell_2 \langle \text{string} \rangle; \epsilon) = (\text{inj}_{\text{string}^*} \{[\]\}^v; \epsilon) \\
&\quad \quad \hookrightarrow \text{IMP}(\text{string}^*; \epsilon) = ([\]; \epsilon) \\
&\quad \hookrightarrow \text{IMP}(\ell_3 \langle \text{string} \rangle^*; \ell_3 \langle s_3 \rangle) = ([\ell_3 \langle s_3 \rangle]; \epsilon) \\
&\quad \quad \hookrightarrow \text{IMP}(\ell_3 \langle \text{string} \rangle; \ell_3 \langle s_3 \rangle) = (\ell_3 \langle s_3 \rangle; \epsilon) \\
&\quad \quad \hookrightarrow \text{IMP}(\ell_3 \langle \text{string} \rangle^*; \epsilon) = ([\]; \epsilon)
\end{aligned}$$

where $v = \text{string}^* \mid \ell_2 \langle \text{string} \rangle$. It is easy to verify, using the typing rules from Chapter 8, that the expression $\ell_1 \langle s_1 \rangle, \text{inj}_{\text{string}^*} \{[\]\}^v, [\ell_3 \langle s_3 \rangle]$ has type σ . \square

9.4 Correctness of Algorithm IMP

If a document A validates a DTD type σ then we expect $\text{IMP}(\sigma; A) = (D; B)$ where D is an expression of type σ , i.e. the judgement $\emptyset \vdash D : \sigma$ is derivable using the typing rules from Chapter 8, and B the empty document ϵ . This is precisely what we define as the *correctness* of

Algorithm IMP.

It is worth pointing out that correctness implies (i) that the expression returned satisfies the aforementioned property and (ii) that Algorithm IMP does not diverge because it always returns a pair of the form $(D; B)$.

First, it is easy to verify that Algorithm IMP implements a total function. In other words, that there is a rule for each possible pair of the form $(\sigma; A)$. This follows from the fact that the pattern A_1, \dots, A_m with $m \geq 1$ matches any sequences of documents, including ϵ (cf. Section 9.3), and that there is a rule for each possible type σ . Second, it is equally easy to verify that Algorithm IMP always converges because it only recurs on simpler cases (cf. Lemma 9.4.1).

Lemma 9.4.1 (Termination of Algorithm IMP). *There exists no XML document A and type σ for which $IMP(\sigma; A)$ diverges.* □

Proof. Let $|\sigma|$ and $|A|$ denote the sizes of σ and A , respectively. A simple analysis of Algorithm IMP shows that the measure $|\sigma| + |A|$ is strictly decreasing in every recursive call. Since $|\sigma| + |A|$ is non-negative, this implies that the algorithm must terminate. □

In order to prove the correctness of Algorithm IMP it is necessary to introduce the notion of a *context*. By keeping track of the context of a type, we will be able to establish an invariant of the computation carried out by the algorithm. A *context with one hole* is defined as follows:

$$\begin{array}{ll}
 \mathcal{C}, \mathcal{C}' ::= \square & \text{(Hole)} \\
 \quad | \mathcal{C}* & \text{(List Context)} \\
 \quad | \mathcal{C}, \tau_1, \dots, \tau_n \quad (n \geq 1) & \text{(Tuple Context)} \\
 \quad | \tau_1 \mid \dots \mid \mathcal{C} \mid \dots \mid \tau_n \quad (n \geq 1) & \text{(Union Context)}
 \end{array}$$

Notice that there is no context that corresponds to the type $\ell \langle \tau \rangle$, and also that a tuple context forbids a subcontext to appear anywhere but in the leftmost position. Given a context \mathcal{C} and

a type τ we write $\mathcal{C}[\tau]$ for the type that results after replacing \square by τ in \mathcal{C} . Similarly, we can replace \square by a context \mathcal{C}' in \mathcal{C} to obtain a new context $\mathcal{C}[\mathcal{C}']$.

Lemma 9.4.2. *For every type τ and every context \mathcal{C} we have $\text{first}(\tau) \subseteq \text{first}(\mathcal{C}[\tau])$. □*

Proof. By induction on the structure of \mathcal{C} .

($\mathcal{C} = \square$) Immediate as $\mathcal{C}[\tau] = \tau$.

($\mathcal{C} = \mathcal{C}'*$) By induction hypothesis $\text{first}(\tau) \subseteq \text{first}(\mathcal{C}'[\tau])$. By definition of $\text{first}()$ it must be $\text{first}(\mathcal{C}'[\tau]) \subseteq \text{first}(\mathcal{C}'[\tau]*) = \text{first}(\mathcal{C}'*[\tau])$. Hence, we conclude that $\text{first}(\tau) \subseteq \text{first}(\mathcal{C}'*[\tau])$.

($\mathcal{C} = \mathcal{C}', \tau_1, \dots, \tau_n$) By induction hypothesis $\text{first}(\tau) \subseteq \text{first}(\mathcal{C}'[\tau])$. By definition of $\text{first}()$ we have $\text{first}(\mathcal{C}'[\tau]) \subseteq \text{first}(\mathcal{C}'[\tau], \tau_1, \dots, \tau_n) = \text{first}((\mathcal{C}', \tau_1, \dots, \tau_n)[\tau])$. Hence, we conclude that $\text{first}(\tau) \subseteq \text{first}((\mathcal{C}', \tau_1, \dots, \tau_n)[\tau])$.

($\mathcal{C} = \tau_1 \mid \dots \mid \mathcal{C}' \mid \dots \mid \tau_n$) By induction hypothesis $\text{first}(\tau) \subseteq \text{first}(\mathcal{C}'[\tau])$. By definition of $\text{first}()$ we have $\text{first}(\mathcal{C}'[\tau]) \subseteq \text{first}(\tau_1 \mid \dots \mid \mathcal{C}'[\tau] \mid \dots \mid \tau_n) = \text{first}((\tau_1 \mid \dots \mid \mathcal{C}' \mid \dots \mid \tau_n)[\tau])$. Hence, we conclude that $\text{first}(\tau) \subseteq \text{first}((\tau_1 \mid \dots \mid \mathcal{C}' \mid \dots \mid \tau_n)[\tau])$. □

Let us now define Algorithm IMP' as a simple extension to Algorithm IMP in which a third argument, the context, is supplied. Algorithm IMP' is shown in Figure 9.2. Algorithm IMP and Algorithm IMP' implement the exact same function; the context passed as an additional argument to the latter is needed exclusively for the correctness proof that follows.

Lemma 9.4.3 (Correctness of Algorithm IMP'). *Let A_1, \dots, A_m with $m \geq 1$ be an XML document in canonical form and σ a well-formed DTD type.*

Hypothesis:

$\text{IMP}'(\text{string}; A_1, \dots, A_m; \mathcal{C})$	$= (A_1; A_2, \dots, A_m)$
$\text{IMP}'(\ell \langle \tau \rangle; A_1, \dots, A_m; \mathcal{C})$	$= \text{let } \ell \langle A' \rangle = A_1$ $(D; B) = \text{IMP}'(\tau; A'; \square)$ in $(\ell \langle D \rangle; A_2, \dots, A_m)$
$\text{IMP}'(\tau^*; A_1, \dots, A_m; \mathcal{C})$	$= \text{if } A_1, \dots, A_m = \epsilon \text{ or } \text{first}(A_1) \not\subseteq \text{first}(\tau) \text{ then}$ $([]; A_1, \dots, A_m)$ else $\text{let } (D_1; B_1) = \text{IMP}'(\tau; A_1, \dots, A_m; \mathcal{C}[\square*])$ $(D_2; B_2) = \text{IMP}'(\tau^*; B_1; \mathcal{C})$ in $(D_1 :: D_2; B_2)$ fi
$\text{IMP}'(\tau_1, \dots, \tau_n; A_1, \dots, A_m; \mathcal{C})$	$= \text{if } \text{first}(A_1) \not\subseteq \text{first}(\tau_1) \text{ then}$ $\text{let } (D_1; B_1) = \text{IMP}'(\tau_1; \epsilon; \square)$ $(D_2; B_2) = \text{IMP}'(\tau_2, \dots, \tau_n; A_1, \dots, A_m; \mathcal{C})$ in $(D_1, D_2; B_2)$ else $\text{let } (D_1; B_1) = \text{IMP}'(\tau_1; A_1, \dots, A_m; \mathcal{C}[\square, \tau_2, \dots, \tau_n])$ $(D_2; B_2) = \text{IMP}'(\tau_2, \dots, \tau_n; B_1; \mathcal{C})$ in $(D_1, D_2; B_2)$ fi
$\text{IMP}'(\tau_1 \mid \dots \mid \tau_n; A_1, \dots, A_m; \mathcal{C})$	$= \text{let } j \in \{h \in 1..n \mid \text{first}(A_1) \subseteq \text{first}(\tau_h)\}$ $(D; B) = \text{IMP}'(\tau_j; A_1, \dots, A_m; \tau_1 \mid \dots \mid \square_j \mid \dots \mid \tau_n)$ $v = \tau_1 \mid \dots \mid \tau_n$ in $(\text{inj}_{\tau_j} \{D\}^v; B)$

Figure 9.2. Algorithm IMP'.

1. Let \mathcal{C} be a context such that A_1, \dots, A_m validates $\mathcal{C}[\sigma]$,
2. There exists a $k \in 0..m$ such that A_1, \dots, A_k validates σ and $A_1, \dots, A_{k'}$ does not validate σ for every $k' \in k+1..m$.

Conclusion:

1. $\text{IMP}'(\sigma; A_1, \dots, A_m; \mathcal{C}) = (D; A_{k+1}, \dots, A_m)$ for some expression D ,

2. $\emptyset \vdash D : \sigma$. □

Proof. By induction on the structure of σ . If $A_1, \dots, A_m = \epsilon$, i.e. if $A_1 = \epsilon$ and $m = 1$, then we assume that $k = 1$; taking $k = 0$ leads to the same case because $A_1, \dots, A_0 = \epsilon$.

($\sigma = \text{string}$) By assumption, there exists a maximal $k \in 0..m$ such that A_1, \dots, A_k validates string. Since for $k = 0$ we have $A_1, \dots, A_k = \epsilon$ and ϵ does not validate string, it follows from the definition of DTD validation that $k = 1$. Hence, by definition of Algorithm IMP' we have $\text{IMP}'(\text{string}; A_1, \dots, A_m; \mathcal{C}) = (A_1; A_2, \dots, A_m)$ and $\emptyset \vdash A_1 : \text{string}$.

($\sigma = \ell \langle \tau \rangle$) By assumption, there exists a maximal $k \in 0..m$ such that A_1, \dots, A_k validates $\ell \langle \tau \rangle$. Since for $k = 0$ we have $A_1, \dots, A_k = \epsilon$ and ϵ does not validate $\ell \langle \tau \rangle$, it follows from the definition of DTD validation that $k = 1$. Hence, it must be $A_1 = \ell \langle A' \rangle$ with A' validating τ . By induction hypothesis we have $\text{IMP}'(\tau; A'; \square) = (D; \epsilon)$ and $\emptyset \vdash D : \tau$, and from the last judgement using (DExp Label) we conclude $\emptyset \vdash \ell \langle D \rangle : \ell \langle \tau \rangle$.

($\sigma = \tau^*$) By assumption there exists a maximal $k \in 0..m$ such that A_1, \dots, A_k validates τ^* . Let us consider three different subcases:

($A_1, \dots, A_m = \epsilon$) Then, we have $A_1 = \epsilon$ and $m = k = 1$. It follows from the definition of Algorithm IMP' that $\text{IMP}'(\tau^*; A_1, \dots, A_m; \mathcal{C}) = ([]; A_1, \dots, A_m)$ and $\emptyset \vdash [] : \tau^*$.

($A_1, \dots, A_m \neq \epsilon$ **and** $k = 0$) Let us show that in this case $\text{first}(A_1) \not\subseteq \text{first}(\tau)$. Suppose that $\text{first}(A_1) \subseteq \text{first}(\tau)$. Because k is maximal, there exists no $k' \in 1..m$ such that $A_1, \dots, A_{k'}$ validates τ^* . Therefore, since A_1, \dots, A_m validates $\mathcal{C}[\tau^*]$, this type must have a subtype of the form $\mathcal{C}'[\tau^*], \tau'$ for some context \mathcal{C}' and type τ' such that $\text{nullable}(\mathcal{C}'[\tau^*]) = \text{true}$ and A_1, \dots, A_h validates τ' for some $h \in 1..m$. Thus, it must

be $\text{first}(A_1) \subseteq \text{first}(\tau')$ by Lemma 9.2.5. But then, using Lemma 9.4.2 and the assumption, we have $\text{first}(A_1) \subseteq \text{first}(\tau) \subseteq \text{first}(\tau^*) \subseteq \mathcal{C}'[\tau^*]$ and $\text{first}(A_1) \subseteq \text{first}(\tau')$, which means that $\text{first}(\mathcal{C}'[\tau^*]) \cap \text{first}(\tau') \neq \emptyset$ so the type $\mathcal{C}'[\tau^*], \tau'$ is not well formed. Contradiction, so it must be $\text{first}(A_1) \not\subseteq \text{first}(\tau)$. By definition of Algorithm IMP' we have $\text{IMP}'(\tau^*; A_1, \dots, A_m; \mathcal{C}) = ([]; A_1, \dots, A_m)$ and $\emptyset \vdash [] : \tau^*$.

$(A_1, \dots, A_m \neq \epsilon \text{ and } k \geq 1)$ Since A_1, \dots, A_k validates τ^* then by Lemma 9.2.5 we have $\text{first}(A_1, \dots, A_k) = \text{first}(A_1) \subseteq \text{first}(\tau^*) = \{b\} \cup \text{first}(\tau)$. Because A_1, \dots, A_k is in canonical form then it must be $\text{first}(A_1) \subseteq \text{first}(\tau)$. Since A_1, \dots, A_k validates τ^* then, from the definition of DTD validation, there exists a maximal $h \in 1..k$ such that A_1, \dots, A_h validates τ and A_{h+1}, \dots, A_k validates τ^* . Additionally, if A_1, \dots, A_m validates $\mathcal{C}[\tau^*]$ then it follows that A_1, \dots, A_m validates $(\mathcal{C}[\square^*])[\tau]$ and also that A_{h+1}, \dots, A_m validates $\mathcal{C}[\tau^*]$. By induction hypothesis $\text{IMP}'(\tau; A_1, \dots, A_m; \mathcal{C}[\square^*]) = (D_1; A_{h+1}, \dots, A_m)$ and $\emptyset \vdash D_1 : \tau$ and $\text{IMP}'(\tau^*; A_{h+1}, \dots, A_m; \mathcal{C}) = (D_2; A_{k+1}, \dots, A_m)$ and $\emptyset \vdash D_2 : \tau^*$. Hence, by (DExp Cons) we conclude that $\emptyset \vdash D_1 :: D_2 : \tau^*$.

$(\sigma = \tau_1, \dots, \tau_n)$ It follows from Lemma 9.2.4 that $A_1, \dots, A_m \neq \epsilon$ and that $k \geq 1$. Let us consider two different subcases:

$(\text{first}(A_1) \not\subseteq \text{first}(\tau_1))$ By Lemma 9.2.5 (contrapositive) there exists no $h \in 1..k$ such that A_1, \dots, A_h validates τ_1 . Therefore, it must be that ϵ validates τ_1 and A_1, \dots, A_k validates τ_2, \dots, τ_n and also A_1, \dots, A_m validates $\mathcal{C}[\tau_2, \dots, \tau_n]$. By induction hypothesis, we have $\text{IMP}'(\tau_1; \epsilon; \square) = (D_1; B_1)$ and $\emptyset \vdash D_1 : \tau_1$ and $\text{IMP}'(\tau_2, \dots, \tau_n; A_1, \dots, A_m; \mathcal{C}) = (D_2; B_2)$ and $\emptyset \vdash D_2 : \tau_2, \dots, \tau_n$. Hence, by (DExp Tuple) we conclude that $\emptyset \vdash D_1, D_2 : \tau_1, \dots, \tau_n$.

($\text{first}(A_1) \subseteq \text{first}(\tau_1)$) Let us show that in this case there exists an $h \in 1..k$ such that A_1, \dots, A_h validates τ_1 . If no such h exists then, because A_1, \dots, A_k validates τ_1, \dots, τ_n , it must be that ϵ validates τ_1 and A_1, \dots, A_m validates τ_2, \dots, τ_n . But then, $\text{first}(A_1) \subseteq \text{first}(\tau_2)$ and $\text{first}(A_1) \subseteq \text{first}(\tau_1)$ so $\text{first}(\tau_1) \cap \text{first}(\tau_2) \neq \emptyset$, which implies that τ_1, \dots, τ_n is not well formed. Therefore, h as defined above must exist. Moreover, since A_1, \dots, A_m validates $\mathcal{C}[\tau_1, \dots, \tau_n]$ then A_1, \dots, A_m validates $(\mathcal{C}[\square, \tau_2, \dots, \tau_n])[\tau_1]$ and A_{h+1}, \dots, A_m validates $\mathcal{C}[\tau_2, \dots, \tau_n]$. By induction hypothesis, we have $\emptyset \vdash D_1 : \tau_1$ and $\text{IMP}'(\tau_1; A_1, \dots, A_m; \mathcal{C}[\square, \tau_2, \dots, \tau_n]) = (D_1; A_{h+1}, \dots, A_m)$ and also that $\text{IMP}'(\tau_2, \dots, \tau_n; A_{h+1}, \dots, A_m; \mathcal{C}) = (D_2; A_{k+1}, \dots, A_m)$ and $\emptyset \vdash D_2 : \tau_2, \dots, \tau_n$. Hence, by (DExp Tuple) we conclude that $\emptyset \vdash D_1, D_2 : \tau_1, \dots, \tau_n$.

($\sigma = \tau_1 \mid \dots \mid \tau_n$) By assumption, context \mathcal{C} is such that A_1, \dots, A_m validates $\mathcal{C}[\tau_1 \mid \dots \mid \tau_n]$ and there exists a maximal $k \in 0..m$ such that A_1, \dots, A_k validates $\tau_1 \mid \dots \mid \tau_n$. Let us consider two different subcases:

($k = 0$) Since ϵ validates $\tau_1 \mid \dots \mid \tau_n$ then by Lemma 9.2.5 we get $\text{first}(\epsilon) \subseteq \text{first}(\tau_1 \mid \dots \mid \tau_n)$. Because $\tau_1 \mid \dots \mid \tau_n$ is well formed there must exist a unique $h \in 1..n$ such that ϵ validates τ_h and $\text{first}(\epsilon) \subseteq \text{first}(\tau_h)$. From the definition of Algorithm IMP' we have $h = j$. Clearly, if A_1, \dots, A_m validates $\mathcal{C}[\tau_1 \mid \dots \mid \tau_n]$ then it follows that A_1, \dots, A_m validates $(\mathcal{C}[\tau_1 \mid \dots \mid \square_j \mid \dots \mid \tau_n])[\tau_j]$, so using the induction hypothesis we get that $\text{IMP}'(\tau_j; A_1, \dots, A_m; \mathcal{C}[\tau_1 \mid \dots \mid \square_j \mid \dots \mid \tau_n]) = (D; A_1, \dots, A_m)$ and $\emptyset \vdash D : \tau_j$ for some expression D . Hence, by (DExp Inj) we conclude that $\emptyset \vdash \text{inj}_{\tau_j} \{D\}^\sigma : \sigma$.

($k \geq 1$) Since A_1, \dots, A_k validates $\tau_1 \mid \dots \mid \tau_n$ then by Lemma 9.2.5 $\text{first}(A_1, \dots, A_k) = \text{first}(A_1) \subseteq \text{first}(\tau_1 \mid \dots \mid \tau_n)$. Because $\tau_1 \mid \dots \mid \tau_n$ is well formed there must ex-

ist a unique $h \in 1..n$ such that A_1, \dots, A_k validates τ_h and $\text{first}(A_1) \subseteq \text{first}(\tau_h)$. From the definition of Algorithm IMP' we have $h = j$. Clearly, if A_1, \dots, A_m validates $\mathcal{C}[\tau_1 \mid \dots \mid \tau_n]$ then A_1, \dots, A_m validates $(\mathcal{C}[\tau_1 \mid \dots \mid \square_j \mid \dots \mid \tau_n])[\tau_j]$, so using the induction hypothesis we get $\text{IMP}'(\tau_j; A_1, \dots, A_m; \mathcal{C}[\tau_1 \mid \dots \mid \square_j \mid \dots \mid \tau_n]) = (D; A_{k+1}, \dots, A_m)$ and $\emptyset \vdash D : \tau_j$ for some expression D . Hence, by (DExp Inj) we conclude that $\emptyset \vdash \text{inj}_{\tau_j}\{D\}^\sigma : \sigma$.

□

Theorem 9.4.4 (Correctness of Algorithm IMP). *Let A_1, \dots, A_m with $m \geq 1$ be an XML document in canonical form and σ a well-formed DTD type. If A_1, \dots, A_m validates the type σ then $\text{IMP}(\sigma; A_1, \dots, A_m) = (D; \epsilon)$ and $\emptyset \vdash D : \sigma$.*

□

Proof. A consequence of Lemma 9.4.3 where $\mathcal{C} = \square$ and $k = m$.

□

9.5 From the $\delta\lambda$ -Calculus to XML

In the preceding sections we have presented an algorithm to “import” XML documents into the $\delta\lambda$ -Calculus so that they can be manipulated in a statically-typed manner. It is conceivable to think about the converse, i.e. an algorithm to “export” $\delta\lambda$ -Calculus values into XML documents. Fortunately, this operation is considerably simpler than its dual, as it basically amounts to erasing all the annotations that exist in the $\delta\lambda$ -Calculus for the purpose of type checking.

In this section we present Algorithm EXP which takes a value in the $\delta\lambda$ -Calculus and returns an XML document (i.e. an element of XDoc). Only a subset of DVal, the set of values in the $\delta\lambda$ -Calculus, can be exported into XML as there is no standard representation for things like variables or abstractions. We refer to this set as XDVal.

$\text{EXP}(s)$	$=$	s
$\text{EXP}([])$	$=$	ϵ
$\text{EXP}(V :: V')$	$=$	if $V' = []$ then $\text{EXP}(V)$ else $\text{EXP}(V), \text{EXP}(V')$ fi
$\text{EXP}(V_1, \dots, V_n)$	$=$	$\text{EXP}(V_1), \dots, \text{EXP}(V_n)$
$\text{EXP}(\ell \langle V \rangle)$	$=$	$\ell \langle \text{EXP}(V) \rangle$
$\text{EXP}(\text{inj}_\tau \{V\}^v)$	$=$	$\text{EXP}(V)$

Figure 9.3. Algorithm EXP.

$V, V' \in \text{XDVal} ::=$	s	(String)
	$[]$	(Nil)
	$V :: V'$	(Cons)
	V_1, \dots, V_n	(Tuple)
	$\ell \langle V \rangle$	(Label)
	$\text{inj}_\tau \{V\}^v$	(Injection)

Algorithm EXP is shown in Figure 9.3. Aside from the second, third and sixth case, the rest of the cases are trivial because they are simple homomorphisms. The case for $V :: V'$ ensures that a list of the form $[V_1, \dots, V_n]$ is translated into $\text{EXP}(V_1), \dots, \text{EXP}(V_n)$ instead of $\text{EXP}(V_1), \dots, \text{EXP}(V_n), \epsilon$.

EXAMPLE 9.5.1. Let V be the value $\ell_1 \langle s_1 \rangle, \text{inj}_{\text{string}^*} \{[]\}^v, [\ell_3 \langle s_3 \rangle]$ with $v = \text{string}^* \mid \ell_2 \langle \text{string} \rangle$ and $\sigma = \ell_1 \langle \text{string} \rangle, v, \ell_3 \langle \text{string} \rangle^*$ (cf. Example 9.3.1). It is easy to verify that $\emptyset \vdash V : \sigma$ and that $\text{EXP}(V) = \ell_1 \langle s_1 \rangle, \epsilon, \ell_3 \langle s_3 \rangle \in \llbracket \sigma \rrbracket$. □

Theorem 9.5.2. Let V be an element of XDVal and σ a DTD type such that $\emptyset \vdash V : \sigma$. Then, it follows that $\text{EXP}(V)$ validates σ . □

Proof. We prove by induction on the structure of V that $\text{EXP}(V) \in \llbracket \sigma \rrbracket$. Clearly, this implies

that $\text{EXP}(V)$ validates σ . We elaborate the second, third and sixth case in Figure 9.3; the other cases are immediate.

$(V = [])$ It follows that $\sigma = \tau^*$ for some type τ . Then, we have $\text{EXP}([]) = \epsilon \in \llbracket \tau^* \rrbracket$.

$(V = V'::V'')$ It follows from (DExp Cons) that $\sigma = \tau^*$ for some type τ , and that $\emptyset \vdash V' : \tau$ and $\emptyset \vdash V'' : \tau^*$. By induction hypothesis, we have $\text{EXP}(V') \in \llbracket \tau \rrbracket$ and $\text{EXP}(V'') \in \llbracket \tau^* \rrbracket$. If $V'' = []$ then $\text{EXP}(V') \in \llbracket \tau \rrbracket \subseteq \llbracket \tau^* \rrbracket$. If $V'' \neq []$ then $\text{EXP}(V'')$ must be of the form A_1, \dots, A_n with $n \geq 1$ and $A_i \in \llbracket \tau \rrbracket$ for $i \in 1..n$. Since $\text{EXP}(V') \in \llbracket \tau \rrbracket$ it must be $\text{EXP}(V'), \text{EXP}(V'') \in \llbracket \tau^* \rrbracket$.

$(V = \text{inj}_{\tau_j} \{V'\}^v)$ It follows from (DExp Inj) that $\emptyset \vdash V' : \tau_j$ with $v = \tau_1 \mid \dots \mid \tau_n$ and $j \in 1..n$.

By induction hypothesis, we have $\text{EXP}(V') \in \llbracket \tau_j \rrbracket \subset \bigcup_{i=1}^n \llbracket \tau_i \rrbracket = \llbracket v \rrbracket$. □

Notice that XML documents returned by Algorithm EXP are not necessarily in canonical form. However, given a value $V \in \text{XDVal}$, we can always obtain an XML document in canonical form by simply erasing the ϵ 's that occur in every tuple of $\text{EXP}(V)$.

Chapter 10

Recursive Types

As discussed in Chapter 6, almost every conceivable data structure can be easily represented in the form of an XML document. Because the $\delta\lambda$ -Calculus is statically typed, however, there are certain expressions that, despite being well behaved, are not typable in the system presented in Chapter 8. This is, in fact, a common phenomenon in type theory: the use of a type system gives us the ability to prove desirable properties like Subject Reduction at the expense of rejecting a subset of the well-behaved programs.

Using the type system from Chapter 8, we can write programs that generate XML documents whose *width* is potentially unbounded but whose *depth* is always fixed. For example, the type $\ell\langle\text{string}^*\rangle$ denotes a set of documents of unbounded width, namely, all documents of the form $\ell\langle s_1, \dots, s_n \rangle$ with $n \geq 0$. But, what is the type of a document that has an unbounded number of nested ℓ 's? Specifically, what is the type of a document like $\ell\langle\ell\langle\dots\ell\langle s \rangle\dots\rangle\rangle$?

In this chapter we shall extend the type system presented in Chapter 8 with recursive types. We will also show how XML documents (and expressions generating XML documents) such as the one shown above can be typed in the extended system.

10.1 Recursive Types

We let X, Y range over an infinite set of type variables TVar . The set DExpTypeUR , which extends DExpTypeU from Chapter 8, is defined inductively by the following grammar:

$\sigma, \tau \in \text{DExpTypeUR} ::=$	X	(Type Variable)
	<code>bool</code>	(Boolean Type)
	<code>string</code>	(String Type)
	$\ell \langle \tau \rangle$	(Label Type)
	τ^*	(List Type)
	τ_1, \dots, τ_n	(Tuple Type)
	$\tau_1 \mid \dots \mid \tau_n$	(Union Type)
	$\tau \rightarrow \sigma$	(Arrow Type)
	$\mu(X).\sigma$	(Recursive Type)

If σ is a type in DExpTypeUR , then we write $\text{ftv}(\sigma)$ for the set of free type variables in σ , $\text{btv}(\sigma)$ for the set of bound type variables in σ and $\text{tv}(\sigma)$ for the set $\text{ftv}(\sigma) \cup \text{btv}(\sigma)$. To avoid confusion, we use the symbol “ \triangleq ” to introduce new definitions (i.e. macros) and reserve the symbol “ $=$ ” for type equations.

A *regular tree* is a tree that has finitely many non-isomorphic subtrees. It follows that every finite tree is a regular tree, but there are infinite trees that are also regular. A recursive type of the form $\mu(X).\sigma$ in which $X \in \text{ftv}(\sigma)$ is a finite representation of an infinite (but regular) tree. For example, the type $\mu(X).(\text{int} \rightarrow X)$ represents an infinite tree that has two different subtrees, namely, `int` and itself. Even though there are infinite trees that are not regular (so-called irregular) we will use the terms “infinite” and “regular” interchangeably.

There are many, in fact infinitely many, finite representations for the same regular tree. Therefore, we need to define a relation by which two recursive types are equated iff they represent the same regular tree. The formalization of this relation is based on that in [AC93]; we

will assume without proof that the main results from that report hold in our system as well.¹

As a first attempt, let us define the relation \approx over elements of DExpTypeUR as a congruence based on the following axiom:

$$\mu(X).\sigma \approx \sigma\{X := \mu(X).\sigma\} \quad (\text{FoldUnfold})$$

This congruence relates types like $\mu(X).l\langle X \rangle$ and $l\langle l\langle \mu(X).l\langle X \rangle \rangle \rangle$ (both of which can be infinitely unfolded as $l\langle l\langle l\langle \dots \rangle \rangle \rangle$) because by (Fold-Unfold) we have (i) $\mu(X).l\langle X \rangle \approx l\langle \mu(X).l\langle X \rangle \rangle$ and (ii) $l\langle \mu(X).l\langle X \rangle \rangle \approx l\langle l\langle \mu(X).l\langle X \rangle \rangle \rangle$, which implies that the former must be congruent to the latter by transitivity.

Unfortunately, as explained in [AC93], this way of defining the congruence \approx results in a relation that is sound but not complete. It is sound because it only relates recursive types that represent the same regular tree, but incomplete because there are types that represent same regular tree that are not related. For example, consider the following definitions:

$$\begin{aligned} \sigma_1 &\triangleq \mu(X).l\langle X \rangle \\ \sigma_2 &\triangleq \mu(X).l\langle l\langle X \rangle \rangle \end{aligned}$$

It is easy to check that the infinite unfolding of both σ_1 and σ_2 is $l\langle l\langle l\langle \dots \rangle \rangle \rangle$. However, and in contrast to the our previous example, there is no type σ_3 such that $\sigma_1 \approx \sigma_3$ and $\sigma_3 \approx \sigma_2$ that would let us conclude the congruency of these two types.

A recursive type $\mu(X).\sigma$ can be thought of as a solution to an equation of the form $X = \sigma$, because the types that result after replacing X by $\mu(X).\sigma$ on both sides of the equality are related by \approx . It is possible to show that non-trivial type equations (these exclude equations such as $X = X$) always have *unique* solutions (see [AC93]). Therefore, if two syntactically

¹For example, we will assume the axiomatization of our congruence (to be presented shortly) is sound and complete with respect to a model for recursive types such as the one presented in that report.

(Refl) $\frac{}{\vdash \sigma \approx \sigma}$	(Symm) $\frac{\vdash \sigma \approx \tau}{\vdash \tau \approx \sigma}$	(Trans) $\frac{\vdash \sigma \approx \tau' \quad \vdash \tau' \approx \tau}{\vdash \sigma \approx \tau}$
(CongLabel) $\frac{\vdash \sigma \approx \tau}{\vdash l \langle \sigma \rangle \approx l \langle \tau \rangle}$	(CongList) $\frac{\vdash \sigma \approx \tau}{\vdash \sigma * \approx \tau *}$	(CongTuple) $\frac{\vdash \sigma_i \approx \tau_i \quad \forall i \in 1..n}{\vdash \sigma_1, \dots, \sigma_n \approx \tau_1, \dots, \tau_n}$
(CongArrow) $\frac{\vdash \sigma \approx \sigma' \quad \vdash \tau \approx \tau'}{\vdash \sigma \rightarrow \tau \approx \sigma' \rightarrow \tau'}$	(CongRec) $\frac{\vdash \sigma \{X := X'\} \approx \tau \{Y := X'\} \quad X' \notin \text{tv}(\sigma) \cup \text{tv}(\tau)}{\vdash \mu(X). \sigma \approx \mu(Y). \tau}$	
(CongUnion) $\frac{\vdash \sigma_i \approx \tau_i \quad \forall i \in 1..n}{\vdash \sigma_1 \mid \dots \mid \sigma_n \approx \tau_1 \mid \dots \mid \tau_n}$	(Contract) $\frac{\vdash \sigma \approx \tau' \{X := \sigma\} \quad \vdash \tau \approx \tau' \{X := \tau\} \quad \tau' \downarrow X}{\vdash \sigma \approx \tau}$	
(FoldUnfold) $\frac{}{\vdash \mu(X). \sigma \approx \sigma \{X := \mu(X). \sigma\}}$	(\approx) $\frac{E \vdash D : \sigma \quad \vdash \sigma \approx \tau}{E \vdash D : \tau}$	

Figure 10.1. Axiomatization of \approx .

different types are solutions to the same equation, they must represent the same regular tree. This observation gives us a powerful method to check if two recursive types, such as σ_1 and σ_2 , are equal: find a type τ with $X \in \text{ftv}(\tau)$ for which $\sigma_1 \approx \tau \{X := \sigma_1\}$ and $\sigma_2 \approx \tau \{X := \sigma_2\}$. For σ_1 and σ_2 as defined above, for example, it suffices to take $\tau \triangleq l \langle l \langle X \rangle \rangle$.

The complete axiomatization of \approx is shown in Figure 10.1.² A type τ is contractive in X , written $\tau \downarrow X$, iff the equation $X = \tau$ has a unique solution. Despite the apparent complexity of this axiomatization, in particular of the rule (Contract), it is a simple task to adapt the (sound

²The axiomatization presented in this figure is not necessarily the most economical because it includes rules that can be derived from other rules. We opted for the explicit inclusion of these non-primitive rules because they are referred in some of the proofs that follow.

and complete) algorithm in [AC93] to decide our congruence.

EXAMPLE 10.1.1. In HTML, we can define nested (unordered) lists using the tag `ul` and items within these lists using the tag `li`.³ In the absence of recursive types, it is only possible to type expressions where the number of nested `ul`'s is statically bounded. That is, it is impossible to assign a type, for example, to a function returning a list whose depth depends on one of its arguments.

Using recursive types, on the other hand, we can define the type of unbounded nested lists as follows:

$$\sigma_1 \triangleq \mu(X). \text{ul} \langle (\text{li} \langle \text{string} \rangle \mid X)^* \rangle$$

This type says that an unordered list is delimited by a `ul` tag whose children are any number of list items or unordered lists. Hence, this type denotes documents that are potentially unbounded both in width and in depth. An example of a term having type σ is:

$$D_1 = \text{ul} \langle [\text{inj}_{\sigma_2} \{\text{li} \langle s_1 \rangle\}^{\sigma_1}, \text{inj}_{\sigma_1} \{\text{ul} \langle [\text{inj}_{\sigma_2} \{\text{li} \langle s_2 \rangle\}^{\sigma_1}] \rangle\}^{\sigma_1}] \rangle$$

where $\sigma_2 \triangleq \text{li} \langle \text{string} \rangle$ and $s_i \in \text{Str}$ for $i \in 1..3$. The judgement $\emptyset \vdash D_1 : \sigma_1$ is derived as follows:

$$\begin{array}{r}
\emptyset \vdash D_1 : \sigma_1 \qquad \qquad \qquad (\approx) \\
\uparrow \emptyset \vdash D_1 : \text{ul} \langle (\sigma_2 \mid \sigma_1)^* \rangle \qquad \qquad \qquad (\text{DExp Label}) \\
\uparrow \emptyset \vdash [\text{inj}_{\sigma_2} \{\text{li} \langle s_1 \rangle\}^{\sigma_1}, \text{inj}_{\sigma_1} \{\text{ul} \langle [\text{inj}_{\sigma_2} \{\text{li} \langle s_2 \rangle\}^{\sigma_1}] \rangle\}^{\sigma_1}] : (\sigma_2 \mid \sigma_1)^* \qquad \qquad \qquad (\text{DExp Cons}) \\
\uparrow \emptyset \vdash \text{inj}_{\sigma_2} \{\text{li} \langle s_1 \rangle\}^{\sigma_1} : \sigma_2 \mid \sigma_1 \qquad \qquad \qquad (\text{DExp Inj}) \\
\uparrow \emptyset \vdash \text{li} \langle s_1 \rangle : \sigma_2 \qquad \qquad \qquad (\text{DExp Label}) \\
\uparrow \dots \\
\uparrow \emptyset \vdash [\text{inj}_{\sigma_1} \{\text{ul} \langle [\text{inj}_{\sigma_2} \{\text{li} \langle s_2 \rangle\}^{\sigma_1}] \rangle\}^{\sigma_1}] : (\sigma_2 \mid \sigma_1)^* \qquad \qquad \qquad (\text{DExp Cons})
\end{array}$$

³In HTML, the term “unordered” is used to refer to lists that are not numbered. Lists are ordered by definition, so the term unordered is really a misnomer.

$$\begin{array}{l}
\uparrow \emptyset \vdash \text{inj}_{\sigma_1} \{ \text{ul} \langle [\text{inj}_{\sigma_2} \{ \text{li} \langle s_2 \rangle \}^{\sigma_1}] \rangle \}^{\sigma_1} : \sigma_2 \mid \sigma_1 \quad (\text{DExp Inj}) \\
\uparrow \emptyset \vdash \text{ul} \langle [\text{inj}_{\sigma_2} \{ \text{li} \langle s_2 \rangle \}^{\sigma_1}] \rangle : \sigma_1 \quad (\approx) \\
\uparrow \emptyset \vdash \text{ul} \langle [\text{inj}_{\sigma_2} \{ \text{li} \langle s_2 \rangle \}^{\sigma_1}] \rangle : \text{ul} \langle (\sigma_2 \mid \sigma_1)^* \rangle \quad (\text{DExp Label}) \\
\uparrow \dots \\
\uparrow \vdash \sigma_1 \approx \text{ul} \langle (\sigma_2 \mid \sigma_1)^* \rangle \quad (\text{FoldUnfold}) \\
\uparrow \emptyset \vdash [] : (\sigma_2 \mid \sigma_1)^* \quad (\text{DExp Nil}) \\
\uparrow \emptyset \vdash \diamond \quad (\text{DEnv } \emptyset) \\
\uparrow \vdash \sigma_1 \approx \text{ul} \langle (\sigma_2 \mid \sigma_1)^* \rangle \quad (\text{FoldUnfold})
\end{array}$$

By analyzing this derivation, it is to see that unordered lists of any depth can be proven to have type σ_1 . The number of times σ_1 needs to be unfolded in a derivation is proportional to the depth of the document in question. \square

EXAMPLE 10.1.2. In this example we show how to write a function F that takes a list of strings and returns a document whose depth equals the length of the list, and whose content is simply the concatenation of all the strings in the list. For example, if the input list is $['x', 'm', 'l']$ then:

$$F ['x', 'm', 'l'] = \ell \langle \ell \langle \ell \langle \text{'xml'} \rangle \rangle \rangle$$

The return type of function F must be a recursive type because the depth of the document that is constructed cannot be determined statically. If the list is empty, then we assume that F returns the empty string. Let $\sigma_2 \triangleq \text{string}$ and $\sigma_1 \triangleq \mu(X).(\sigma_2 \mid \ell \langle X \rangle)$ in:

$$\begin{array}{l}
F : \sigma_2^* \rightarrow \sigma_1 = \\
(\lambda x : \sigma_2^*. \text{letrec } G : \sigma_2^* \rightarrow \sigma_2 \rightarrow \sigma_1 = \\
\quad \lambda x : \sigma_2^*. \lambda y : \sigma_2. \text{if null}(x) \text{ then } \text{inj}_{\sigma_2} \{y\}^{\sigma_1} \\
\quad \quad \quad \text{else } \text{inj}_{\sigma_1} \{ \ell \langle G(x.\text{tl}) \rangle (y \oplus x.\text{hd}) \}^{\sigma_1} \\
\text{in} \\
G \ x \ \text{' '})
\end{array}$$

It is easy to verify that the definition of F agrees with the specification given above; the only difference, compared to the example shown earlier, is that the document returned contains a number of injections that are needed for the code to type check. \square

An unfortunate consequence of having added the rule (\approx) to the type system from Chapter 8, is that the resulting system is no longer *syntax directed*. That is, in principle, it is possible to apply the rule (\approx) to the same subterm occurrence more than once, even though this is not necessary. Stated differently, given an expression D for which $E \vdash D : \sigma$ holds for some fixed E and σ , the derivation that proves this judgement is no longer unique. However, as shown by the following lemma, it is possible to re-arrange a derivation so that the rule (\approx) is applied exactly once for every subterm occurrence.

Lemma 10.1.3. *Every type derivation can be re-arranged into a standard type derivation in which exactly one instance of the rule (\approx) is applied to every subterm occurrence.* \square

Proof. This result follows from the fact that the relation \approx is reflexive and transitive. By reflexivity, we can insert an instance of (\approx) with the premise $\vdash \tau \approx \tau$, where τ is the type of the conclusion, between instances of two rules that are not (\approx). By transitivity, it is always possible to collapse $n \geq 2$ instances of (\approx) into one. \square

The Subject Reduction result from Chapter 8 can be easily adjusted for the system with recursive types. The proof is similar to that of Theorem 8.4.2, except for the treatment of the rule (\approx).

Theorem 10.1.4 (Subject Reduction). *Let D and D' be document expressions and τ an element of DExpTypeUR . If $D \longrightarrow D'$ and $E \vdash D : \tau$ then $E \vdash D' : \tau$.* \square

Proof. By induction on derivations using the rules from Figures 7.1, 7.2, 7.3 and 8.2. By Lemma 10.1.3 we can assume that the derivation of the judgement $E \vdash D : \tau$ is *standard*. Due to the similarity to the proof of Theorem 8.4.2, only a few selected cases are shown.

(Red Head) By assumption we have $(V :: V') \cdot \text{hd} \longrightarrow V$ and $E \vdash (V :: V') \cdot \text{hd} : \tau$. Then, it follows by (\approx) that $E \vdash (V :: V') \cdot \text{hd} : \tau'$ and $\vdash \tau' \approx \tau$, and by (DExp Head) that $E \vdash (V :: V') : \tau'^*$. Thus, by (\approx) once again we have $E \vdash (V :: V') : \tau''^*$ with $\vdash \tau''^* \approx \tau'^*$, and by (DExp Cons) that $E \vdash V : \tau''$. Therefore, by (CongList) and (Trans) it must be $\vdash \tau'' \approx \tau$, so from the last judgement and the rule (\approx) we conclude that $E \vdash V : \tau$ as desired.

(Red Sel) By assumption we have $(\ell \langle V \rangle) / \ell \longrightarrow V$ and $E \vdash (\ell \langle V \rangle) / \ell : \tau$. Then, it follows by (\approx) that $E \vdash (\ell \langle V \rangle) / \ell : \tau'$ and $\vdash \tau' \approx \tau$, and by (DExp Sel) that $E \vdash \ell \langle V \rangle : \ell \langle \tau' \rangle$. Thus, by (\approx) once again we have $E \vdash \ell \langle V \rangle : \ell \langle \tau'' \rangle$ with $\vdash \ell \langle \tau'' \rangle \approx \ell \langle \tau' \rangle$, and by (DExp Label) that $E \vdash V : \tau''$. Therefore, by (CongLabel) and (Trans) it must be $\vdash \tau'' \approx \tau$, so from the last judgement and the rule (\approx) we conclude that $E \vdash V : \tau$ as desired.

(Red DHead) By assumption we have $D \cdot \text{hd} \longrightarrow D' \cdot \text{hd}$ because $D \longrightarrow D'$ and $E \vdash D \cdot \text{hd} : \tau$. Then, it follows by (\approx) that $E \vdash D \cdot \text{hd} : \tau'$ with $\vdash \tau' \approx \tau$, and by (DExp Head) that $E \vdash D : \tau'^*$. By induction hypothesis we have $E \vdash D' : \tau'^*$. From $\vdash \tau' \approx \tau$ and (CongList) we have $\vdash \tau'^* \approx \tau^*$, so from (\approx) we get $E \vdash D' : \tau^*$. Therefore, by (DExp Head) we conclude that $E \vdash D' \cdot \text{hd} : \tau$ as desired.

(Red DSel) By assumption we have $D / \ell \longrightarrow D' / \ell$ because $D \longrightarrow D'$ and $E \vdash D / \ell : \tau$. Then, it follows by (\approx) that $E \vdash D / \ell : \tau'$ with $\vdash \tau' \approx \tau$, and by (DExp Sel) that $E \vdash D : \ell \langle \tau' \rangle$. By induction hypothesis we have $E \vdash D' : \ell \langle \tau' \rangle$. From $\vdash \tau' \approx \tau$ and (CongLabel) we have $\vdash \ell \langle \tau' \rangle \approx \ell \langle \tau \rangle$, so from (\approx) we get $E \vdash D' : \ell \langle \tau \rangle$. Therefore, by (DExp Sel) we conclude that $E \vdash D' / \ell : \tau$ as desired.

(Red Beta) By assumption we have $(\lambda x : \sigma. D) V \longrightarrow D \{x := V\}$ and $E \vdash (\lambda x : \sigma. D) V : \tau$. Then, it follows by (\approx) that $E \vdash (\lambda x : \sigma. D) V : \tau'$ with $\vdash \tau' \approx \tau$, and by (DExp App) that

$E \vdash (\lambda x : \sigma. D) : \sigma \rightarrow \tau'$ and $E \vdash V : \sigma$. Hence, by (DExp Abs) we get $E, x : \sigma \vdash D : \tau'$ and, from the last two judgements using Lemma 8.4.1, we have $E \vdash D\{x := V\} : \tau'$. Therefore, by (\approx) using $\vdash \tau' \approx \tau$ we conclude that $E \vdash D\{x := V\} : \tau$ as desired.

(Red AppLeft) By assumption we have $DD'' \longrightarrow D'D''$ because $D \longrightarrow D'$ and $E \vdash DD'' : \tau$.

Then, it follows by (\approx) that $E \vdash DD'' : \tau'$ with $\vdash \tau' \approx \tau$, and by (DExp App) that $E \vdash D : \sigma \rightarrow \tau'$ and $E \vdash D'' : \sigma$. By induction hypothesis we have $E \vdash D' : \sigma \rightarrow \tau'$, so by (DExp App) we get $E \vdash D'D'' : \tau'$. Therefore, by (\approx) using $\vdash \tau' \approx \tau$ we conclude that $E \vdash D'D'' : \tau$ as desired.

(Red AppRight) Similar to (Red AppLeft). □

Chapter 11

Conclusions to Part II

In this part we presented a statically-typed calculus for manipulating XML documents that we called $\delta\lambda$ -Calculus. This calculus was defined as the union of two separate calculi: the δ -Calculus, with primitives to manipulate XML documents; and the λ -Calculus, with primitives to define abstractions over XML documents.

We also proposed a fairly rich type system for our calculus that employs union types and recursive types, among others. Subject Reduction for this system is proved in Theorem 8.4.2 and Theorem 10.1.4.

11.1 Discussion and Future Work

An interesting extension to the type system for the $\delta\lambda$ -Calculus is the addition of a subtyping relation. In the absence of subtyping, for example, a document passed to a function must have a type identical to the input type of the function, which is often very restrictive. Clearly, a document that has *at least* the structure expected by a function is a good candidate to be an

argument. The use of a subtyping relation, together with a subsumption rule, is a natural way of lifting this restriction.

We can exploit the fact that there are no side effects in the $\delta\lambda$ -Calculus to define a relation where subtyping can take place both in width and in depth (cf. discussion in Section 5.1). Such a subtyping relation is defined by the following rules:

$$\begin{array}{c}
\text{(Env } \emptyset\text{)} \\
\frac{}{\emptyset \vdash \diamond}
\end{array}
\qquad
\begin{array}{c}
\text{(EnvAdd)} \\
\frac{E \vdash \diamond \quad (X \leq Y) \notin \text{dom}(E)}{E, X \leq Y \vdash \diamond}
\end{array}
\qquad
\begin{array}{c}
\text{(EnvGet)} \\
\frac{E, X \leq Y, E' \vdash \diamond}{E, X \leq Y, E' \vdash X \leq Y}
\end{array}$$

$$\begin{array}{c}
\text{(SubGround)} \\
\frac{E \vdash \diamond \quad \tau \in \{\text{bool}, \text{string}\}}{E \vdash \tau \leq \tau}
\end{array}
\qquad
\begin{array}{c}
\text{(SubLabel)} \\
\frac{E \vdash \tau \leq \sigma}{E \vdash \ell \langle \tau \rangle \leq \ell \langle \sigma \rangle}
\end{array}
\qquad
\begin{array}{c}
\text{(SubList)} \\
\frac{E \vdash \tau \leq \sigma}{E \vdash \tau * \leq \sigma *}
\end{array}$$

$$\begin{array}{c}
\text{(SubTuple)} \\
\frac{E \vdash \tau_i \leq \sigma_i \quad \forall i \in 1..m \quad m \leq n}{E \vdash \tau_1, \dots, \tau_n \leq \sigma_1, \dots, \sigma_m}
\end{array}
\qquad
\begin{array}{c}
\text{(SubArrow)} \\
\frac{E \vdash \tau' \leq \tau \quad E \vdash \sigma \leq \sigma'}{E \vdash \tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'}
\end{array}$$

$$\begin{array}{c}
\text{(SubUnion)} \\
\frac{E \vdash \tau_i \leq \sigma_i \quad \forall i \in 1..n \quad n \leq m}{E \vdash \tau_1 \mid \dots \mid \tau_n \leq \sigma_1 \mid \dots \mid \sigma_m}
\end{array}
\qquad
\begin{array}{c}
\text{(SubRec)} \\
\frac{E, X \leq Y \vdash \tau \leq \sigma}{E \vdash \mu(X).\tau \leq \mu(Y).\sigma}
\end{array}$$

Notice that in (SubTuple) we have $m \leq n$ and in (SubUnion) we have $n \leq m$. That is, using (SubTuple) we can assign to a tuple the type of any of its prefixes, while using (SubUnion) we can lose precision by adding more cases to a union type.

The subtyping relation described above is not as powerful as the one proposed in [HP01] for XDuce. In that report, the authors present a subtyping relation based on the denotation of types; more precisely, they state that $\vdash \sigma \leq \tau$ iff $\llbracket \sigma \rrbracket \subseteq \llbracket \tau \rrbracket$ for an appropriate definition of $\llbracket \cdot \rrbracket$ (akin to our definition in Section 9.1). An advantage of this definition is the ability to relate types of different kinds, e.g. lists and tuples, which is clearly impossible with the rules shown above. A disadvantage, on the other hand, is that the resulting algorithm for their relation can be shown

to be DEXPTIME-complete.

Future work includes the complete development of a type system for the $\delta\lambda$ -Calculus extended with subtyping, as well as a more precise comparison to the system in [HP01].

Chapter 12

The DL Language

As a “proof of concept” we have implemented an interpreter for a language based on the $\delta\lambda$ -Calculus. We refer to this language as DL. This chapter describes the syntax of DL as well as the semantics of a few constructs that are not present in the basic calculus.

12.1 Syntax

Following a current trend (in languages such as XSLT, XML Schemas, etc.), we have opted for an XML-based syntax for the language DL. Thus, every DL expression is a legal XML document.

The advantages of this approach are the following:

- Expressions (programs) can be embedded as part of XML documents. For example, an XML document can carry an “embedded stylesheet”, which is a piece of DL code that translates the contents of the document to a displayable format such as HTML.
- Off-the-shelf tools, such as validating parsers, can be used to implement compilers/interpreters for DL.

- Programmable text editors (e.g., Emacs or XEmacs) can be easily extended to aid programmers write syntactically correct code.

Based on our experience, the only disadvantage of using an XML-based syntax is the fact that programs tend to be long due to the verbosity of the constructs. However, to a certain extent, this problem was offset by the use of an extension to our editor of choice (XEmacs) to help in the creation/edition of DL programs.

As in other languages whose syntax is XML-based, we employ XML namespaces [Worc] to distinguish active vs. pasive data. More specifically, every expression `<<expr>>` presented to our interpreter is evaluated in the following context:

```
<?xml version="1.0"?>
<dl:top-level xmlns:dl="http://types.bu.edu/xml/dlcalculus"
              xmlns:ty="http://types.bu.edu/xml/dlcalculus/types">
  <<expr>>
</dl:top-level>
```

which adds the XML header `<?xml version="1.0"?>`, so that a standard XML parser recognizes the document as parsable, and also defines the namespaces `dl` and `ty`. Therefore, every sub-expression in `<<expr>>` of the form `<dl: . . . >` or `<ty: . . . >` will be recognized as a construct in DL as opposed to pasive data (as suggested by their names, the prefix `dl` is reserved for terms and the prefix `ty` is reserved for types).

The following meta-variables are used throughout this chapter for the presentation of the syntax:

- `s, s1, . . .` range over an infinite set of strings;
- `l, l1, . . .` range over an infinite set of labels;
- `x, y, z, f, g, t, t1, . . .` range over an infinite set of variable names;

- n, m range over the set of natural numbers.

The set of strings, labels and variable names coincide with the set of text nodes, element names and attribute values, respectively, as defined in the W3C Recommendation for XML [Wora].¹

12.2 Strings and Labeled Expressions

Given an alphabet Σ , a string in the $\delta\lambda$ -Calculus was written as ' w ' for $w \in \Sigma^*$. Strings in the DL language define the *contents* of a document. Thus, everything that is not *markup* is a string or, using XML terminology, a text node.² Although not necessary in general, it is also possible to define a string using the construct `<dl:string>`. For example, the following two expressions are identical:

1. This is an e-mail message.
2. `<dl:string>This is an e-mail message.</dl:string>`

The expression `<dl:string>` is often needed for readability —specially when the string spans multiple lines— and to define the empty string ϵ as `<dl:string/>`.

Labeled expression in the $\delta\lambda$ -Calculus are written as $\ell \langle D \rangle$ for some expression D and label ℓ . In the DL language, we use the more standard (and more verbose!) syntax where D is delimited by `< ℓ >` and `</ ℓ >`. To exemplify, the e-mail message from Example 6.2 is written as:

```
<email>
  <sender>santiago@cs.bu.edu</sender>
  <receiver>all@cs.bu.edu</receiver>
  <subject>An e-mail message as an XML document</subject>
  <message>This is an e-mail message.</message>
</email>
```

¹This is a byproduct of the use of a standard XML parser to syntactically analyze DL expressions.

²Recall that we exclude from our framework things like processing instructions, attributes, etc.

Strings and labeled expressions are evaluated in accordance to the operational semantics defined in Section 7.1 for the $\delta\lambda$ -Calculus. That is, a string evaluates to itself and a labeled expression evaluates to a labeled expression (of the same label) where its sub-expression is fully evaluated.

12.3 Tuples

Tuples in the $\delta\lambda$ -Calculus are written as D_1, \dots, D_n ; thus each component of a tuple is separated by a “,”. In DL, following standard XML syntax, tuples can be implicit. That is, tuples can be defined simply by writing a sequence of expressions one after the other with no delimiters or separators. For example, the expression

```
<sender>santiago@cs.bu.edu</sender>
<receiver>all@cs.bu.edu</receiver>
This is an e-mail message.
```

is a tuple of length 3 where the first two components are labeled expressions and the last component is a string. The BNF syntax for tuples of expressions (a similar rule is used for tuples of types) is defined by the following production:

```
<<dl-tuple>> ::= <<dl-expr>> | <<dl-expr>> <<dl-tuple>>
```

It is also possible (and often necessary) to define a tuple explicitly. The expression `<dl:tuple>` can be used for that purpose. For example, the explicit tuple that corresponds to the implicit tuple shown above is

```
<dl:tuple>
  <sender>santiago@cs.bu.edu</sender>
  <receiver>all@cs.bu.edu</receiver>
  This is an e-mail message.
</dl:tuple>
```

Regardless of whether a tuple is implicit or explicit, its components can be selected using `<dl:project>`. For example, the expression

```

<dl:project index="2">
  <dl:tuple>
    <sender>santiago@cs.bu.edu</sender>
    <receiver>all@cs.bu.edu</receiver>
    This is an e-mail message.
  </dl:tuple>
</dl:project>

```

evaluates to `<receiver>all@cs.bu.edu</receiver>`. The projection index is specified by means of the (mandatory) attribute `index`, whose value must be a number greater or equal to 1.

12.4 Lists

As in the $\delta\lambda$ -Calculus, lists in DL are defined using the standard constructors “nil” and “cons”.

The syntax for these constructors is the following:

```

<dl:nil/>          <dl:cons>
                   <<dl-tuple>>
                   <<dl-expr>>
                   </dl:cons>

```

Note that the first argument to `<dl:cons>` is (implicitly) a tuple. That is, everything but the last sub-expression of `<dl:cons>` is inserted into the list. For example, the following expression evaluates to a list whose only element is a tuple of two labeled expressions:

```

<dl:cons>
  <sender>santiago@cs.bu.edu</sender>
  <receiver>all@cs.bu.edu</receiver>
  <dl:nil/>
</dl:cons>

```

It is also possible to use `<dl:list>` as a constructor, but this is just syntactic sugar for an expression containing nested `<dl:cons>`, in the same way $[D_1, \dots, D_n]$ is sugar for $D_1 :: \dots (D_n :: [])$ in the $\delta\lambda$ -Calculus. For example, the expression

```

<dl:list>
  <sender>santiago@cs.bu.edu</sender>
  <sender>all@cs.bu.edu</sender>
</dl:list>

```

defines a list of two elements both of which are labeled expressions. Notice that the `<dl:list>` constructor does not group any sub-expressions into tuples; this effect can be achieved by the *explicit* use of `<dl:tuple>`.

Lists in DL can be accessed using the standard destructors “head” and “tail”. The syntax for these destructors is the following:

```
<dl:head>                <dl:tail>
  <<dl-expr>>             <<dl-expr>>
</dl:head>               </dl:tail>
```

It is worth pointing out that the sub-expression of both `<dl:head>` and `<dl:tail>` is not allowed to be a tuple, as such an expression will inevitably generate a type error (or a runtime error in a dynamically-typed framework). In accordance to the operational semantics defined for the $\delta\lambda$ -Calculus, a list must be fully evaluated before its head or its tail can be accessed.

12.5 Variables

Variables can be either global or local. The former are defined using `<dl:define>` and the latter using `<dl:let>`. The syntax of these expressions is shown next:

```
<dl:define name="x">      <dl:let name="x">
  <<dl-tuple>>             <<dl-tuple>>
</dl:define>             <dl:in/>
                          <<dl-tuple>>
                          </dl:let>
```

A `<dl:define>` expression evaluates to the value of its sub-expression, binding that value to the variable defined by the attribute `name` as a side effect.

The value of a variable named “x” can be accessed by writing `<dl:variable name="x"/>`. Since a variable is one of the base cases of our syntax, a `<dl:variable>` expression in DL is required to be an empty XML element (as indicated by the use of “/”).

12.6 Path Expressions

The expressions `<dl:select>`, `<dl:project>`, `<dl:head>` and `<dl:tail>` can be regarded as document *destructors*. Due to the inherent structure of XML documents, DL expressions encountered in practice often contain sub-expressions where these destructors appear nested. For example, suppose “D” is defined as follows:

```
<dl:define name="D">
  <email>
    <sender>santiago@cs.bu.edu</sender>
    <dl:list>
      <receiver>all@cs.bu.edu</receiver>
      <receiver>santiago@cs.bu.edu</receiver>
    </dl:list>
  </email>
</dl:define>
```

In order to select the e-mail address of the first receiver, we need an expression that traverses the tree denoted by D from the root and returns the second leaf. Namely,

```
<dl:select label="receiver">
  <dl:head>
    <dl:project index="2">
      <dl:select label="email">
        <dl:variable name="D"/>
      </dl:select>
    </dl:project>
  </dl:head>
</dl:select>
```

Since expressions such as the one shown above are common in DL, a special (more succinct) syntax is also provided. Specifically, the last expression can also be written as `<dl:expr value="D/email.2.hd/receiver"/>`. The *path* expression "D/email.2.hd/receiver" is intended to be read from left to right: first select email, then project the second component, then take the head and finally select receiver. The syntax of path expressions is defined in Section 12.12.

12.7 Functions

As in the $\delta\lambda$ -Calculus, DL provides ways of defining and applying lambda abstractions. The syntax for these two constructs is the following:

```
<dl:abstraction param="x" param-type="t">      <dl:application>
  <<dl-tuple>>                                  <<dl-expr>> <<dl-tuple>>
</dl:abstraction>                              </dl:application>
```

The attributes `param` and `param-type` are required in `<dl:abstraction>`. These two attributes are used, respectively, to declare the name and the type of the variable being bound. Recall that `t` ranges over a set of variable names; the reader is referred to Section 12.8 for further details on how these type variables are declared.

A `<dl:application>` expression expects at least two sub-expressions. Notice that, according to the syntax shown above, all but the leftmost sub-expression of `<dl:application>` are regarded as an *implicit* tuple. For example, for an appropriate definition of the type variable “`t`”, the following expression shows how the identity function is applied to a binary tuple.

```
<dl:application>
  <dl:abstraction param="x" param-type="t">
    <dl:variable name="x"/>
  </dl:abstraction>
  <email>santiago@cs.bu.edu</email>
  <email>all@cs.bu.edu</email>
</dl:application>
```

Since, in practice, abstractions are often given names, DL provides special constructs for defining named and applying named functions. The syntax for these two constructs is the following:

```
<dl:function name="f" param="x" param-type="t"> <dl:function-call name="f">
  <<dl-tuple>>                                  <<dl-tuple>>
</dl:function>                                  </dl:function-call>
```

An additional advantage of the use of `<dl:function>` over `<dl:abstraction>` is that the former allows for definitions to be recursive. In fact, the syntax of `<dl:function>` shown above can be regarded as sugar for

```

<dl:define name="f">
  <dl:fix-point param="f" param-type="t'">
    <dl:abstraction param="x" param-type="t">
      <<dl-tuple>>
    </dl:abstraction>
  </dl:fix-point>
</dl:define>

```

where `<dl:fix-point>` is the DL construct that corresponds to $(\text{fix } f:\sigma.D)$ in the $\delta\lambda$ -Calculus, and where “t'” denotes an arrow type whose domain is “t” and whose codomain is the type of `<<dl-tuple>>`.

12.8 Types

The type system of DL is based on that presented in Chapter 8 for the $\delta\lambda$ -Calculus. In addition to the type constructors presented in that chapter, DL includes (i) type variables and (ii) a `<ty:void/>`. The reader is referred to Section 12.12 for the complete syntax of all the type constructors available in DL.

As in the case of expressions, a sequence of types with no separators or delimiters defines an *implicit* tuple type. For example, the following type denotes lists whose components are binary tuples of labeled expressions:

```

<ty:list>
  <ty:email><ty:string/></ty:email>
  <ty:email><ty:string/></ty:email>
</ty:list>

```

Using the syntax from Chapter 8, this type would be written as $(\text{email}\langle\text{string}\rangle, \text{email}\langle\text{string}\rangle)^*$.

Notice that the syntax of labeled types is `<ty:email>...</ty:email>` instead of the more intuitive syntax `<email>...</email>`. Thus, the label of a labeled type also belongs the namespace defined by the prefix `ty`, a requirement that simplifies the task of the parser.³

³An unfortunate consequence of this decision is that the keywords `string`, `list`, etc. cannot be used as la-

Type variables can be globally defined using the `<ty:define>` construct. Type variables play an important role in DL since they are needed to define the value of the `param-type` attribute in expressions such as `<dl:abstraction>` and `<dl:function>`. For example, the following fragment of DL code shows how to define a type variable and then a function whose domain is the type assigned to the variable.

```
<ty:define name="Email">
  <ty:email><ty:string/></ty:email>
</ty:define>

<dl:abstraction param="x" param-type="Email">
  <dl:expr value="x/email"/>
</dl:abstraction>
```

Types in DL can also be evaluated. Every type constructor that is not a type variable evaluates to itself; bound type variables evaluate to the types introduced via `<ty:define>`. Thus, as evaluation is inductive on the structure of types, it is possible to define types using type variables.

To exemplify, the list type shown above can also be written as

```
<ty:list>
  <ty:variable name="Email"/>
  <ty:variable name="Email"/>
</ty:list>
```

provided the type variable `Email` is defined as in the last example. The evaluation of a type fails only if an unbound type variable is referenced via `<ty:variable>`; the type of every type expression is `<ty:kind/>`.

There is an alternative way of assigning a name to a type: every element that has `ty` as prefix also accepts an *optional* attribute `name`. This feature is particularly useful when defining union types since the subcomponents of these types must be named as well (cf. Section 12.9). As an example, let's define the type `Recipient` as follows:

beled types. This, however, can be remedied by introducing a more general (and more verbose) construct such as `<ty:label name="l">`.

```

<ty:union name="Recipient">
  <ty:email name="Email">
    <ty:string/>
  </ty:email>
  <ty:newsgroup name="Newsgroup">
    <ty:string/>
  </ty:newsgroup>
</ty:union>

```

12.9 Injections and Cases

Injections and Cases in DL are similar in structure to their $\delta\lambda$ -Calculus counterparts, except that instead of using types we use type variables —as in other typed constructs like `<dl:abstraction>` and `<dl:function>`— to annotate the terms. The syntax for these two constructs is shown next.

```

<dl:inject type="x" name="t">
  <<dl-tuple>>
</dl:inject>

<dl:case bind="x">
  <<dl-expr>>
  <dl:match name="t1">
    <<dl-tuple>>
  </dl:match>
  ...
  <dl:match name="tn">
    <<dl-tuple>>
  </dl:match>
</dl:case>

```

The notation “...” that is part of the syntax of `<dl:case>` indicates that the number of `<dl:match>`’s is unbounded. In fact, the number of `<dl:match>`’s must coincide with the number of types in the union type in question (cf. Figure 8.1).

12.10 Miscellaneous

DL provides a few additional constructs that are not part of (and cannot be de-sugared into) the underlying calculus. In this section we cover `<dl:load>` and `<dl:display>`.

The `<dl:load>` construct is used to load definitions into the interpreter's environment. The syntax is `<dl:load filename="<<file-name>>" />`. Expressions in `<<file-name>>` are evaluated in order, and the result of the evaluation is `<dl:void/>` (and, hence, the type of a `<dl:load>` is `<ty:void/>`).

The `<dl:display>` construct is used to pretty print the value of an expression. The syntax of this expression is:

```
<dl:display>
  <<dl-tuple>>
</dl:display>
```

As `<dl:load>`, the evaluation of `<dl:display>` produces a side effect, namely, adding one or more characters to the standard output. The result of evaluating a `<dl:display>` is also `<dl:void/>` (and, hence, the type of a `<dl:display>` is also `<ty:void/>`).

12.11 An Example

Let us show how Example 8.2.2 from Section 8.2 can be written using the syntax of DL. We begin by defining the type of an XML e-mail, the type of an HTML e-mail and the types of the functions F and G.

```
<ty:define name="XMLEmail">
  <ty:email>
    <ty:sender><ty:string/></ty:sender>
    <ty:list name="RecsList">
      <ty:receiver><ty:string/></ty:receiver>
    </ty:list>
    <ty:subject><ty:string/></ty:subject>
    <ty:message><ty:string/></ty:message>
  </ty:email>
</ty:define>

<ty:define name="HTMLEmail">
  <ty:html>
```

```

    <ty:head>
      <ty:h3><ty:string/></ty:h3>
    </ty:head>
    <ty:body>
      <ty:list>
        <ty:p><ty:string/></ty:p>
      </ty:list>
    </ty:body>
  </ty:html>
</ty:define>

```

```

<ty:define name="Type-G">
  <ty:arrow>
    <ty:variable name="RecsList"/>
    <ty:string/>
  </ty:arrow>
</ty:define>

```

```

<ty:define name="Type-F">
  <ty:arrow>
    <ty:variable name="XMLEmail"/>
    <ty:variable name="HTMLEmail"/>
  </ty:arrow>
</ty:define>

```

Notice how the type of F and the type of G, respectively Type-F and Type-G, are defined in terms of the type variables XMLEmail, HTMLEmail and RecsList, and also how the latter names a subpart of the type bound to XMLEmail. Given these type definitions, document d1 and functions F and G are defined as follows:

```

<dl:define name="d1">
  <email>
    <sender>santiago@cs.bu.edu</sender>
    <dl:list>
      <receiver>all@cs.bu.edu</receiver>
      <receiver>santiago@cs.bu.edu</receiver>
    </dl:list>
    <subject>An e-mail message as an XML document</subject>
    <message>This is an e-mail message.</message>
  </email>
</dl:define>

<dl:function name="G" param="z" name-type="Type-G">

```

```

<dl:if>
  <dl:null>
    <dl:variable name="z"/>
  </dl:null>
<dl:then/>
  <dl:string/> <!-- null string -->
<dl:else/>
  <dl:concat>
    <dl:expr value="z.hd/receiver"/>
    <dl:function-call name="G">
      <dl:expr value="z.tl"/>
    </dl:function-call>
  </dl:concat>
</dl:if>
</dl:function>

<dl:function name="F" param="x" name-type="Type-F">
  <dl:let name="y">
    <dl:expr value="x/email"/>
  <dl:in/>
  <html>
    <head>
      <h3><dl:expr value="y.3/subject"/></h3>
    </head>
    <body>
      <dl:list>
        <p><dl:expr value="y.1/sender"/></p>
        <p>
          <dl:function-call name="G">
            <dl:expr value="y.2"/>
          </dl:function-call>
        </p>
        <p><dl:expr value="y.4/message"/></p>
      </dl:list>
    </body>
  </html>
</dl:let>
</dl:function>

```

This example also illustrates the use of `<dl:if>` and `<dl:let>`. The `<dl:if>` expression is used in conjunction with the empty DL expressions `<dl:then/>` and `<dl:else/>`, and is evaluated in the standard way (i.e. if the value of its first sub-expression is `<dl:true/>` then the “then” case is evaluated, if it is `<dl:false/>` then the “else” case is evaluated). The evaluation of `<dl:let>`

is also standard, and follows from the way let expressions are de-sugared into applications.

12.12 Complete Syntax

```
<<tuple>> ::= <<dl-tuple>> | <<ty-tuple>>
```

```
<<dl-tuple>> ::= <<dl-expr>> | <<dl-expr>> <<dl-tuple>>
```

```
<<dl-expr>> ::= s
```

```
    | <dl:nil/>
```

```
    | <dl:cons>
      <<dl-tuple>> <<dl-expr>>
    </dl:cons>
```

```
    | <dl:tuple>
      <<dl-tuple>>
    </dl:tuple>
```

```
    | <l>
      <<dl-tuple>>
    </l>
```

```
    | <dl:head>
      <<dl-expr>>
    </dl:head>
```

```
    | <dl:tail>
      <<dl-expr>>
    </dl:tail>
```

```
    | <dl:project index="n">
      <<dl-tuple>>
    </dl:project>
```

```
    | <dl:select label="l">
      <<dl-expr>>
    </dl:select>
```

```
    | <dl:variable name="x"/>
```

```
    | <dl:abstraction param="x" param-type="t">
```

```

    <<dl-tuple>>
  </dl:abstraction>

| <dl:application>
  <<dl-expr>> <<dl-tuple>>
</dl:application>

| <dl:true/>

| <dl:false/>

| <dl:concat>
  <<dl-expr>> <<dl-expr>>
</dl:concat>

| <dl:fix-point param="x" param-type="t">
  <<dl-tuple>>
</dl:concat>

| <dl:if>
  <<dl-expr>>
  <dl:then/>
  <<dl-tuple>>
  <dl:else/>
  <<dl-tuple>>
</dl:if>

| <dl:null>
  <<dl-tuple>>
</dl:null>

| <dl:labeled label="l">
  <<dl-tuple>>
</dl:labeled>

| <dl:inject type="x" name="t">
  <<dl-tuple>>
</dl:inject>

| <dl:case bind="x">
  <<dl-expr>>
  <dl:match name="t1">
    <<dl-tuple>>
  </dl:match>
  ...

```

```

        <dl:match name="tn">
            <<dl-tuple>>
        </dl:match>
    </dl:case>

| <dl:string>s</dl:string>

| <dl:define name="x">
    <<dl-tuple>>
</dl:define>

| <dl:function name="f" param="x" param-type="t">
    <<dl-tuple>>
</dl:function>

| <dl:function-call name="f">
    <<dl-tuple>>
</dl:function-call>

| <dl:expr value="<<path-expr>>" />

| <dl:let name="x">
    <<dl-tuple>>
    <dl:in/>
    <<dl-tuple>>
</dl:let>

| <dl:load filename="s" />

| <dl:list>
    <<dl-tuple>>
</dl:list>

| <dl:display>
    <<dl-tuple>>
<dl:display>

| <dl:void/>

<<path-expr>> ::= x

    | <<path-expr>>.n

    | <<path-expr>>/l

```

```
| <<path-expr>>.hd
| <<path-expr>>.tl
<<ty-tuple>> ::= <<type>> | <<type>> <<ty-tuple>>
<<type>> ::= <ty:void/>
| <ty:bool/>
| <ty:string/>
| <ty:variable name="t"/>
| <ty:l>
  <<ty-tuple>>
</ty:l>
| <ty:list>
  <<ty-tuple>>
</ty:list>
| <ty:tuple>
  <<ty-tuple>>
</ty:tuple>
| <ty:union>
  <<ty-tuple>>
</ty:union>
| <ty:arrow>
  <<ty-tuple>>
  <<type>>
</ty:arrow>
| <ty:define name="t">
  <<ty-tuple>>
</ty:define>
```

A Appendix to Part II

$D, D' \in \text{DExp} ::=$	s	(String)
	$[]$	(Nil)
	$D :: D'$	(Cons)
	$D_1, \dots, D_n \quad (n \geq 2)$	(Tuple)
	$\ell \langle D \rangle$	(Label)
	$D \cdot \text{hd}$	(Head)
	$D \cdot \text{tl}$	(Tail)
	$D \cdot n$	(Projection)
	D / ℓ	(Selection)
	$\text{inj}_\tau \{D\}^v$	(Injection)
	$\text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n$	(Case)
	x	(Variable)
	$\lambda x : \tau. D$	(Abstraction)
	$D D'$	(Application)
	true	(True)
	false	(False)
	$D \oplus D'$	(String Concat)
	$\text{fix } x : \tau. D$	(Fix-point)
	$\text{if } D_1 \text{ then } D_2 \text{ else } D_3$	(IfThenElse)
	$\text{null}(D)$	(Null Predicate)
	$\text{labeled}(D, \ell)$	(Labeled Predicate)
$V, V' \in \text{DVal} ::=$	s	(String)
	true	(True)
	false	(False)
	$[]$	(Nil)
	$V :: V'$	(Cons)
	V_1, \dots, V_n	(Tuple)
	$\ell \langle V \rangle$	(Label)
	$\text{inj}_\tau \{V\}^v$	(Injection)
	x	(Variable)
	$\lambda x : \tau. D$	(Abstraction)

Figure 12.1. Expressions and Values

$(V::V')\cdot\text{hd} \longrightarrow V$	(Red Head)
$(V::V')\cdot\text{tl} \longrightarrow V'$	(Red Tail)
$(V_1, \dots, V_n)\cdot k \longrightarrow V_k \quad k \in 1..n$	(Red Proj)
$(\ell \langle V \rangle) / \ell \longrightarrow V$	(Red Sel)
$D \longrightarrow D' \Rightarrow D\cdot\text{hd} \longrightarrow D'\cdot\text{hd}$	(Red DHead)
$D \longrightarrow D' \Rightarrow D\cdot\text{tl} \longrightarrow D'\cdot\text{tl}$	(Red DTail)
$D \longrightarrow D' \Rightarrow D\cdot k \longrightarrow D'\cdot k$	(Red DProj)
$D \longrightarrow D' \Rightarrow D / \ell \longrightarrow D' / \ell$	(Red DSel)
$D \longrightarrow D' \Rightarrow D::D'' \longrightarrow D'::D''$	(Red ConsHead)
$D \longrightarrow D' \Rightarrow V::D \longrightarrow V::D'$	(Red ConsTail)
$D \longrightarrow D' \Rightarrow \ell \langle D \rangle \longrightarrow \ell \langle D' \rangle$	(Red Label)
$D_k \longrightarrow D'_k \Rightarrow V_1, \dots, V_{k-1}, D_k, \dots, D_n \longrightarrow V_1, \dots, V_{k-1}, D'_k, \dots, D_n$	(Red Tuple)
$(\lambda x:\tau.D)V \longrightarrow D\{x := V\}$	(Red Beta)
$D \longrightarrow D' \Rightarrow DD'' \longrightarrow D'D''$	(Red AppLeft)
$D \longrightarrow D' \Rightarrow VD \longrightarrow VD'$	(Red AppRight)
$\text{fix } x:\tau.D \longrightarrow D\{x := \text{fix } x:\tau.D\}$	(Red Fix)
$V \oplus V' \longrightarrow V'' \quad (V'' = \text{concat}(V, V'))$	(Red Concat)
$D \longrightarrow D' \Rightarrow D \oplus D'' \longrightarrow D' \oplus D''$	(Red ConcatLeft)
$D \longrightarrow D' \Rightarrow V \oplus D \longrightarrow V \oplus D'$	(Red ConcatRight)
$\text{if true then } D_1 \text{ else } D_2 \longrightarrow D_1$	(Red IfTrue)
$\text{if false then } D_1 \text{ else } D_2 \longrightarrow D_2$	(Red IfFalse)
$D \longrightarrow D' \Rightarrow \text{if } D \text{ then } D_1 \text{ else } D_2 \longrightarrow \text{if } D' \text{ then } D_1 \text{ else } D_2$	(Red If)
$\text{null}([\]) \longrightarrow \text{true}$	(Red NullTrue)
$\text{null}(V) \longrightarrow \text{false} \quad (V \neq [\])$	(Red NullFalse)
$D \longrightarrow D' \Rightarrow \text{null}(D) \longrightarrow \text{null}(D')$	(Red Null)
$\text{labeled}(\ell \langle V \rangle, \ell) \longrightarrow \text{true}$	(Red LabeledTrue)
$\text{labeled}(V, \ell) \longrightarrow \text{false} \quad (V \neq \ell \langle V' \rangle)$	(Red LabeledFalse)
$D \longrightarrow D' \Rightarrow \text{labeled}(D, \ell) \longrightarrow \text{labeled}(D', \ell)$	(Red Labeled)
$D \longrightarrow D' \Rightarrow \text{inj}_\tau\{D\}^v \longrightarrow \text{inj}_\tau\{D'\}^v$	(Red Inj)
$\text{case inj}_{\tau_j}\{V\}^v \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow D_j\{y := V\} \quad (j \in 1..n)$	(Red CaseInj)
$D \longrightarrow D' \Rightarrow$ $\text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n \longrightarrow \text{case } D' \text{ bind } y \tau_1 \Rightarrow D_1 \cdots \tau_n \Rightarrow D_n$	(Red Case)

Figure 12.2. Operational Semantics

$\text{(DEnv } \emptyset)$ $\frac{}{\emptyset \vdash \diamond}$		$\text{(DEnv } x)$ $\frac{E \vdash \diamond \quad x \notin \text{dom}(E)}{E, x : \tau \vdash \diamond}$	
(DExp Str) $\frac{E \vdash \diamond}{E \vdash s : \text{string}}$	(DExp Nil) $\frac{E \vdash \diamond}{E \vdash [] : \tau^*}$	(DExp Cons) $\frac{E \vdash D : \tau \quad E \vdash D' : \tau^*}{E \vdash D :: D' : \tau^*}$	
(DExp Head) $\frac{E \vdash D : \tau^*}{E \vdash D.\text{hd} : \tau}$	(DExp Tail) $\frac{E \vdash D : \tau^*}{E \vdash D.\text{tl} : \tau^*}$	(DExp Label) $\frac{E \vdash D : \tau}{E \vdash \ell \langle D \rangle : \ell \langle \tau \rangle}$	
(DExp Sel) $\frac{E \vdash D : \ell \langle \tau \rangle}{E \vdash D/\ell : \tau}$	(DExp Tuple) $\frac{E \vdash D_i : \tau_i \quad \forall i \in 1..n}{E \vdash D_1, \dots, D_n : \tau_1, \dots, \tau_n}$	(DExp Proj) $\frac{E \vdash D : \tau_1, \dots, \tau_n \quad k \in 1..n}{E \vdash D.k : \tau_k}$	
(DExp Var) $\frac{E, x : \tau, E' \vdash \diamond}{E, x : \tau, E' \vdash x : \tau}$	(DExp Abs) $\frac{E, x : \tau \vdash D : \sigma}{E \vdash \lambda x : \tau. D : \tau \rightarrow \sigma}$	(DExp App) $\frac{E \vdash D : \tau \rightarrow \sigma \quad E \vdash D' : \tau}{E \vdash D D' : \sigma}$	
(DExp Bool) $\frac{E \vdash \diamond \quad T \in \{\text{true}, \text{false}\}}{E \vdash T : \text{bool}}$	(DExp Concat) $\frac{E \vdash D : \text{string} \quad E \vdash D' : \text{string}}{E \vdash D \oplus D' : \text{string}}$	(DExp Fix) $\frac{E, x : \tau \vdash D : \tau}{E \vdash \text{fix } x : \tau. D : \tau}$	
(DExp If) $\frac{E \vdash D_1 : \text{bool} \quad E \vdash D_2 : \tau \quad E \vdash D_3 : \tau}{E \vdash \text{if } D_1 \text{ then } D_2 \text{ else } D_3 : \tau}$	(DExp Null) $\frac{E \vdash D : \tau^*}{E \vdash \text{null}(D) : \text{bool}}$	(DExp Labeled) $\frac{E \vdash D : \tau}{E \vdash \text{labeled}(D, \ell) : \text{bool}}$	

Figure 12.3. Typing Rules (1 of 3)

$$\begin{array}{c}
\text{(DExp Inj)} \\
\frac{E \vdash D : \tau_j \quad j \in 1..n}{E \vdash \text{inj}_{\tau_j}\{D\}^v : v} \quad (v = \tau_1 \mid \dots \mid \tau_n) \\
\\
\text{(DExp Case)} \\
\frac{E \vdash D : v \quad E, y : \tau_j \vdash D_j : \sigma \quad \forall j \in 1..n}{E \vdash \text{case } D \text{ bind } y \tau_1 \Rightarrow D_1 \dots \tau_n \Rightarrow D_n : \sigma} \quad (v = \tau_1 \mid \dots \mid \tau_n)
\end{array}$$

Figure 12.4. Typing Rules (2 of 3)

$$\begin{array}{ccc}
\text{(Refl)} & \text{(Symm)} & \text{(Trans)} \\
\frac{}{\vdash \sigma \approx \sigma} & \frac{\vdash \sigma \approx \tau}{\vdash \tau \approx \sigma} & \frac{\vdash \sigma \approx \tau' \quad \vdash \tau' \approx \tau}{\vdash \sigma \approx \tau} \\
\\
\text{(CongLabel)} & \text{(CongList)} & \text{(CongTuple)} \\
\frac{\vdash \sigma \approx \tau}{\vdash l \langle \sigma \rangle \approx l \langle \tau \rangle} & \frac{\vdash \sigma \approx \tau}{\vdash \sigma * \approx \tau *} & \frac{\vdash \sigma_i \approx \tau_i \quad \forall i \in 1..n}{\vdash \sigma_1, \dots, \sigma_n \approx \tau_1, \dots, \tau_n} \\
\\
\text{(CongArrow)} & \text{(CongRec)} & \\
\frac{\vdash \sigma \approx \sigma' \quad \vdash \tau \approx \tau'}{\vdash \sigma \rightarrow \tau \approx \sigma' \rightarrow \tau'} & \frac{\vdash \sigma \{X := X'\} \approx \tau \{Y := X'\} \quad X' \notin \text{tv}(\sigma) \cup \text{tv}(\tau)}{\vdash \mu(X). \sigma \approx \mu(Y). \tau} & \\
\\
\text{(CongUnion)} & \text{(Contract)} & \\
\frac{\vdash \sigma_i \approx \tau_i \quad \forall i \in 1..n}{\vdash \sigma_1 \mid \dots \mid \sigma_n \approx \tau_1 \mid \dots \mid \tau_n} & \frac{\vdash \sigma \approx \tau' \{X := \sigma\} \quad \vdash \tau \approx \tau' \{X := \tau\} \quad \tau' \downarrow X}{\vdash \sigma \approx \tau} & \\
\\
\text{(FoldUnfold)} & (\approx) & \\
\frac{}{\vdash \mu(X). \sigma \approx \sigma \{X := \mu(X). \sigma\}} & \frac{E \vdash D : \sigma \quad \vdash \sigma \approx \tau}{E \vdash D : \tau} &
\end{array}$$

Figure 12.5. Typing Rules (3 of 3)

Part III

XML-Fluent Mobile Agents

Chapter 13

Introduction to Part III

In Parts I and II we presented two calculi, the Channeled Ambient Calculus and the $\delta\lambda$ -Calculus, that were designed for very different purposes. In this part, we will show how these calculi can be combined into a single *two-tier* calculi. The goal is to design a typed language in which computation can be carried out by mobile agents exchanging data in the form of XML documents.

Communication in **CAC** is limited to only a few different kinds of terms that we called expressions; these include ambient names, channel names and (paths of) capabilities. Expressions comprise the lower tier of **CAC**, while the upper tier is comprised by processes that cannot be communicated—in this sense **CAC** is not a higher-order process calculus. A natural way to augment **CAC**, or the Ambient Calculus for that matter, is to extend the set of expressions so that computation can take place not only at the upper tier, but also at this lower tier. An example of such an extension for the Ambient Calculus is outlined in [AKPG01].

Our proposal, similar in spirit to that in [AKPG01], is to augment the lower tier of **CAC** with primitives from the $\delta\lambda$ -Calculus so that processes can exchange, in addition to names and capabilities, data in the form of XML documents. We refer to the new calculus as $\delta\lambda$ -**CAC**.

EXAMPLE 13.0.1. There is a server that knows how to format XML-based e-mail messages into HTML. A client sends an agent (ambient *client_rqst*) to the server to request the formatting of the e-mail message D_1 (as defined in Example 7.1.1). Once the message is formatted, another agent (ambient *client_rply*) returns to the client and outputs the result over channel *html_rply*. The process under consideration is (Server | Client).

```

Server = server[xml_rqst,html_rply;
  open client_rqst.0 |
  xml_rqst(x1).html_rply⟨
    letrec G = (λz.if null(z) then “
      else (z.hd/receiver) ⊕ G(z.tl))
    in
    let F = (λx.let y = x/email in
      html<head<h3<y.3/subject>>,
        body<[p<y.1/sender>,p<G(y.2)>,
          p<y.4/message>]>>)
      in
      F x1⟩.0]

Client = client[html_rply;
  open client_rply.0 |
  html_rply(x2).R |
  client_rqst[;out client.in server.0 |
  xml_rqst⟨D1⟩.html_rply(x3).
  client_rply[;out server.in client.html_rply⟨x3⟩.0]]

```

Notice that the server declares two channels, namely *xml_rqst* and *html_rply*, over which the first agent sends a request and receives a reply, while *client* only declares *html_rply* as only the reply is communicated within the client. As we shall explain in Chapter 14, the $\delta\lambda$ -Calculus expressions (letrec $G = \dots Fx_1$) in the definition of Server and D_1 in the definition of Client, must be evaluated before the respective communications take place. \square

Chapter 14

XML-Fluent Agents

The syntax of $\delta\lambda$ -**CAC**, being an extension of **CAC**, is defined in terms of the set **Proc** (same as in **CAC**) and a new set **Exp**. Let **Exp** be the set generated by the grammar that results after *merging* the grammars for **PExp** (cf. Section 2.1) and for **DExp** (cf. Section 7.1) . Notice that **Exp** properly contains **PExp** \cup **DExp** as, for example, it includes the term $(\lambda x : \tau.x)(in\ n)$ which is not in the union of these two sets. We use M, N and D to range over **Exp** and P, Q and R to range over **Proc**.

The operational semantics of $\delta\lambda$ -**CAC** includes all the reduction rules from **CAC** and all the reduction rules from the $\delta\lambda$ -Calculus, as well as those shown in Figure 14.1. We reserve V to range over the set of values implied by all these rules.

In contrast to **PExp**, there are now expressions in **Exp** that are not values. Because our interest is the communication of values only, we must re-define the rule (Red Comm) to ensure that expressions are evaluated before they are communicated.

$$m[c_i^{i \in I}; c_k \langle V \rangle . R \mid c_k(n : \sigma) . P \mid Q] \longrightarrow m[c_i^{i \in I}; R \mid P\{n := V\} \mid Q] \quad \text{if } k \in I \quad (\text{Red Comm})$$

$M \longrightarrow N \Rightarrow \text{in } M \longrightarrow \text{in } N$	(RedInExp)
$M \longrightarrow N \Rightarrow \text{out } M \longrightarrow \text{out } N$	(RedOutExp)
$M \longrightarrow N \Rightarrow \text{open } M \longrightarrow \text{open } N$	(RedOpenExp)
$M \longrightarrow N \Rightarrow M.M' \longrightarrow N.M'$	(RedPathLeft)
$M \longrightarrow N \Rightarrow V.M \longrightarrow V.N$	(RedPathRight)
$M \longrightarrow N \Rightarrow M\langle M' \rangle.P \longrightarrow N\langle M' \rangle.P$	(RedOutputLeft)
$M \longrightarrow N \Rightarrow V\langle M \rangle.P \longrightarrow V\langle N \rangle.P$	(RedOutputRight)

Figure 14.1. Additional Rules for $\delta\lambda$ -CAC

The only difference between this rule and the one from Figure 2.2 is the use of V instead of M , which forces an expression to be evaluated before the exchange can take place.

EXAMPLE 14.0.2. Consider Example 13.0.1 once again. Let V_1 be the value defined by the following equations:

$$\begin{aligned}
 V_1 &= \text{html}\langle V_2, V_3 \rangle \\
 V_2 &= \text{head}\langle \text{h3}\langle \text{'An e-mail message as an XML document'} \rangle \rangle \\
 V_3 &= \text{body}\langle [\text{p}\langle \text{'santiago@cs.bu.edu'} \rangle, \\
 &\quad \text{p}\langle \text{'all@cs.bu.edusantiago@cs.bu.edu'} \rangle, \\
 &\quad \text{p}\langle \text{'This is an e-mail message.'} \rangle] \rangle
 \end{aligned}$$

Based on the operational semantics of $\delta\lambda$ -CAC we can easily verify that $(\text{Server} \mid \text{Client}) \longrightarrow^{\equiv} \text{server}[\text{xml_rqst}, \text{html_rply}; \mathbf{0}] \mid \text{client}[\text{html_rply}; R\{x_2 := V_1\}]$. □

14.1 Type System

The type system for $\delta\lambda$ -CAC is based on the type system for CAC presented in Chapter 3, and the type system for the $\delta\lambda$ -Calculus presented in Chapters 8 and 10. The sets ProcType and ExpType are defined inductively by the following grammars:

$\rho \in \text{ProcType}$	$::=$	$\text{pro}[c_i : \wedge \tau_i]^{i \in I}$	(Process Type)
$\rho, \sigma, \tau, v \in \text{ExpType}$	$::=$	X	(Type Variable)
		bool	(Boolean Type)
		string	(String Type)
		$\ell \langle \tau \rangle$	(Label Type)
		τ^*	(List Type)
		τ_1, \dots, τ_n	(Tuple Type)
		$\tau_1 \mid \dots \mid \tau_n$	(Union Type)
		$\tau \rightarrow \sigma$	(Arrow Type)
		$\mu(X).\sigma$	(Recursive Type)
		$\wedge \sigma$	(Channel Type)
		$\text{amb}[c_i : \wedge \tau_i]^{i \in I}$	(Ambient Type)
		$\text{cap}[c_i : \wedge \tau_i]^{i \in I}$	(Capability Type)

The typing rules for $\delta\lambda\text{-CAC}$ are those from Figure 3.1 (assigning types to the processes, names and capabilities) and those from Figures 12.3, 12.4 and 12.5 (assigning types to the rest of the expressions). Recall that in Part I, a type environment was defined as a pair $E; F$ where E is a mapping between ambient names and types, and F a mapping between channel names and types. Therefore, in order to be consistent, the typing rules for $\delta\lambda\text{-CAC}$ from Part II (Figures 12.3, 12.4 and 12.5) must be re-defined using $E; F$ instead of just E . Notice that, in addition to the typing rules for processes, only (Env c) and (Exp c) make use of the mapping F .

EXAMPLE 14.1.1. Let us show how to type Example 13.0.1. Define the expression types σ_i for $i \in 1..3$ and τ_1 as follows:

$$\begin{aligned}
\sigma_1 &\triangleq \text{email} \langle \sigma_2 \rangle \\
\sigma_2 &\triangleq \text{sender} \langle \text{string} \rangle, \sigma_3, \text{subject} \langle \text{string} \rangle, \text{message} \langle \text{string} \rangle \\
\sigma_3 &\triangleq \text{receiver} \langle \text{string} \rangle^* \\
\tau_1 &\triangleq \text{html} \langle \text{head} \langle \text{h3} \langle \text{string} \rangle \rangle, \text{body} \langle \text{p} \langle \text{string} \rangle^* \rangle \rangle
\end{aligned}$$

Let Server' and Client' be the type annotated versions of Server and Client where the bound

variables have the following types $x_1 : \sigma_1, x_2 : \tau_1, x_3 : \tau_1, x : \sigma_1, y : \sigma_2$ and $z : \sigma_3$, and where F and G are annotated with the types $\sigma_1 \rightarrow \tau_1$ and $\sigma_3 \rightarrow \text{string}$, respectively. Then, it is easy to verify that the following judgement is derivable:

$$\emptyset; \emptyset \vdash (\nu \text{ server} : \text{amb}[\text{xml_rqst} : \wedge \sigma_1, \text{html_rply} : \wedge \tau_1]). \\ (\nu \text{ client} : \text{amb}[\text{html_rply} : \wedge \tau_1]). (\text{Server}' \mid \text{Client}') : \text{pro}[\]$$

where $\text{pro}[\]$ is a process type $\text{pro}[c_i : \wedge \tau_i]^{i \in I}$ for which $I = \emptyset$. □

Theorem 14.1.2 (Subject Reduction for Expressions). *Let \hat{M} and \hat{N} be expressions. If $\hat{M} \longrightarrow \hat{N}$ and $E; F \vdash \hat{M} : \sigma$ then $E; F \vdash \hat{N} : \sigma$.* □

Proof. The proof for the cases in Figure 14.1 follows directly from the induction hypothesis; the other cases are proven in Theorem 8.4.2. □

Theorem 14.1.3 (Subject Reduction for Processes). *Let \hat{P} and \hat{Q} be processes. If $\hat{P} \longrightarrow \hat{Q}$ and $E; F \vdash \hat{P} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$ then $E; F \vdash \hat{Q} : \text{pro}[c_k : \wedge \tau_k]^{k \in K}$.* □

Proof. A consequence of Theorems 14.1.2 and 3.2.8. □

Chapter 15

Conclusions to Part III

In this part we introduced $\delta\lambda$ -CAC, a calculus based on the calculi presented in the previous two parts. The main results for $\delta\lambda$ -CAC are, in essence, a consequence of the corresponding results for both the $\delta\lambda$ -Calculus and CAC. Thus, the only novelty was the way expressions in the $\delta\lambda$ -Calculus were enriched, giving a programmer the ability to “compute” with expressions as well as with processes.

Although $\delta\lambda$ -CAC is the result of combining the $\delta\lambda$ -Calculus and CAC, the two are essentially kept separate in our framework, in the sense that communication between processes is limited to expressions (values) which, by definition, cannot include processes. We steer clear of a *higher-order* $\delta\lambda$ -CAC, where processes can exchange other processes (in addition to expressions), something that will certainly reproduce many of the challenges already encountered in higher-order versions of the π -calculus, e.g. in the work of Hennessy and his collaborators [YH99, YH00]. Since our interest is the implementation of a language based on $\delta\lambda$ -CAC, there is something to be said in favor of keeping our conceptual framework as simple as possible. —

Bibliography

- [AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Prog. Langs. & Sys.*, 15(4):575–631, 1993.
- [AKPG00] T. Amtoft, A. J. Kfoury, and S. M. Pericas-Geersten. What are polymorphically-typed ambients? Technical Report BUCS-TR-2000-021, Comp. Sci. Dept., Boston Univ., Dec. 2000.
- [AKPG01] T. Amtoft, A. J. Kfoury, and S. M. Pericas-Geertsen. What are polymorphically-typed ambients? In D. Sands, ed., *ESOP 2001, Genova*, vol. 2028 of *LNCS*, pp. 206–220. Springer-Verlag, Apr. 2001. This downloadable extended summary is more detailed than the article in the ESOP proceedings. A full report is [AKPG00].
- [AWL94] A. S. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 163–173, 1994.
- [Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [BPG00a] M. Bugliesi and S. M. Pericas-Geertsen. Depth subtyping and type inference for object calculi. In *Proc. Seventh Workshop on Foundations of Object-Oriented Languages*,

Boston, Mass., U.S.A., 2000.

- [BPG00b] M. Bugliesi and S. M. Pericas-Geertsen. Type inference for variant object types. *Inform. & Comput.*, 2000. To appear.
- [Car99] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, eds., *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *LNCS*, pp. 51–94. Springer-Verlag, 1999.
- [CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, ed., *FoSSaCS'98*, vol. 1378 of *LNCS*, pp. 140–155. Springer-Verlag, 1998.
- [CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL99, San Antonio, Texas*, pp. 79–92. ACM Press, Jan. 1999.
- [Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.
- [FGL⁺96] C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *CONCUR 1996*, vol. 1119 of *LNCS*, pp. 406–421. Springer-Verlag, 1996.
- [GH99] S. Gay and M. Hole. Types and subtypes for client-server interactions. In *Proc. European Symp. on Programming*, vol. 1576 of *LNCS*, pp. 74–90. Springer-Verlag, 1999.
- [HP00] H. Hosoya and B. Pierce. Regular expression types for xml. In *Proc. 2000 Int'l Conf. Functional Programming*. ACM Press, 2000.
- [HP01] H. Hosoya and B. Pierce. Regular expression pattern matching for xml. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, 2001.

- [Mil99] R. Milner. *Communicating and Mobile Systems: The π -Calculus*. Cambridge Press, 1999.
- [RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *POPL 1998*, pp. 378–390. ACM Press, 1998.
- [RH99] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL99, San Antonio, Texas*, pp. 93–104. ACM Press, 1999.
- [SV99] P. Sewell and J. Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop (CSFW-12), Mordano, Italy*, June 1999.
- [VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of *LNCS*. Springer-Verlag, 1999.
- [Wora] World Wide Web Consortium (W3C), <http://www.w3.org/XML/>. *Extensible Markup Language (XML)*.
- [Worb] World Wide Web Consortium (W3C), <http://www.w3.org/XML/XSL.html>. *Extensible Stylesheet Language*.
- [Worc] World Wide Web Consortium (W3C), <http://www.w3.org/TR/1999/REC-xml-names-19990114/>. *Namespaces in XML*.
- [Word] World Wide Web Consortium (W3C), <http://www.w3.org/XML/Query.html>. *XML Query*.
- [Wore] World Wide Web Consortium (W3C), <http://www.w3.org/XML/Schema.html>. *XML Schema*.

- [YH99] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order mobile processes. In *CONCUR 1999*, vol. 1664 of *LNCS*, pp. 557–573. Springer-Verlag, 1999.
- [YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In *LICS 2000*, pp. 334–345, 2000.
- [Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *FOSSACS 2000, Berlin*, vol. 1784 of *LNCS*, pp. 375–390. Springer-Verlag, 2000.