Types and Effects for Secure Service Orchestration

Massimo Bartoletti Pierpaolo Degano Gian Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy

Abstract

A distributed calculus is proposed for describing networks of services. We model service interaction through a call-by-property invocation mechanism, by specifying the security constraints that make their composition safe. A static approach is then proposed to determine how to compose services and guarantee that their execution is always secure, without resorting to any dynamic check.

1 Introduction

The ability of selecting and assembling together heterogeneous services is an important step towards the full development of service-oriented computing [21, 20, 10]. A *service* is a stand-alone component distributed over a network, and made available through standard interaction mechanisms. Orchestration of services may require peculiar mechanisms to handle complex interaction patterns (e.g. to implement transactions), while enforcing non-functional requirements on the system behaviour (e.g. security and service level agreement). Service orchestration heavily depends on which information about a service is made public, on how to choose those services that match the user's requirements, and on their actual run-time behaviour. Security makes service orchestration even harder. Services may be offered by different providers, which only partially trust each other. On the one hand, providers have to guarantee the delivered service to respect a given security policy, in any interaction with the operational environment, and regardless of who actually called the service. On the other hand, clients may want to protect themselves from the services invoked.

A major problem is how to select a *plan* for secure service orchestration. This amounts to selecting from the network those services that accomplish the requested task, while respecting the security constraints on demand. Services that locally obey the property imposed by a request are not always good candidates, because their behaviour may affect security of the whole composition. For example, consider a device with limited computational power that downloads an applet from the network and then delegates a remote service to run it. Although the contract between the device and the code provider is fulfilled, the applet may violate a security policy enforced by the executer. To determine the viable plans, one has to check the effects of all the available applets against the security policies of all the remote executers.

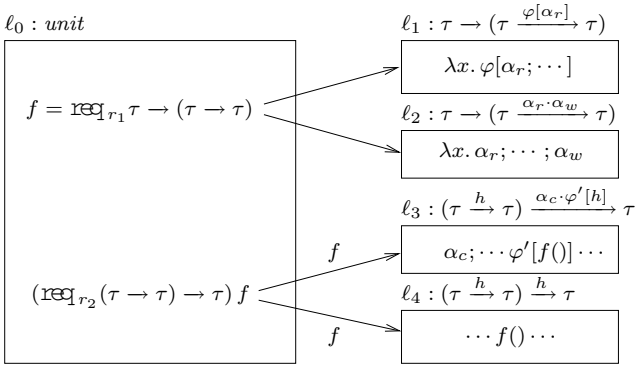
In this paper, we propose a solution to this problem, within a distributed framework. Services are functional units in an enriched λ-calculus, they are explicitly located at network sites, and have a published public interface. Unlike standard syntactic signatures, this interface includes an abstraction of the service behaviour, in the form of annotated types. To obtain a service with a specific behaviour, a client queries the network for a published interface matching the requirements — a sort of *call-by-property* invocation. Security is implemented by wrapping the critical blocks of code inside *security framings* (with possibly nested scopes), that enforce the relevant policies during the execution of the block. In the spirit of history-based security [1], a security policy can inspect the whole history at a given site. Since our framework is fully distributed, our policies cannot span over multiple sites.

We introduce a type and effect system for our calculus [11, 18, 24]. The type of a service describes its I/O behaviour, while the effect, in the form of a *history expression*, represents those aspects of its behaviour relevant to security. History expressions extend regular expressions with information about the selection of services, coupled with their corresponding effect.

Our main result is a way of extracting from a history expression all the *viable* plans, i.e. those that drive secure executions. This is a two-stage construction. A novel transformation of history expressions makes them model-checkable for validity. Valid history expressions guarantee that the services they are extracted from never go wrong at run-time. From valid histories it is then immediate to obtain the viable plans, that make any execution monitor unneeded.

This paper builds over [3], borrowing from it and extending history-based security policies, the call-by-property invocation mechanism, history expressions and the verification technique. Together with the notion of plans, we fur-

ther add here an explicit notion of location and of localized executions, so allowing several clients and services to run concurrently. Our planning technique acts as a *trusted orchestrator* of services, that constructs the plans for a client, by considering the view of the network at the moment the client is injected. The provided plans guarantee that the invoked services always respect the required properties. Thus, in our framework the only trusted entity is the orchestrator, and neither clients nor services need to be such. In particular, the orchestrator infers functional and behavioural types of each service. Also, it is responsible for certifying the service code, for publishing its interface, and for guaranteeing that services will not arbitrarily change their code on-the-fly: when this happens, services need to be certified again. All these extensions bring forth considerable technical complications in the definitions of the operational semantics of the calculus (in Sect. 3), of history expressions (in Sect. 4), of the type and effect system (in Sect. 5). The orchestrator is introduced in Sect. 6. In spite of the added technicalities, the present analysis is significantly finer grained than the all-or-nothing analysis of [3].



2 A motivating example

To illustrate our approach, consider the scenario in the above figure. The boxes model services, distributed over a network. Each box is decorated with the location ℓ_i where the service is published, and with the public interface of the service. This interface is in the form of an annotated type, which is an abstraction of the service behaviour.

The client at site ℓ_0 is a device with limited computational capabilities, wanting to execute some code downloaded from the network. To do that, the client issues two requests in sequence. Its public interface is the singleton $unit$ type, meaning that the client cannot be invoked by anyone. The request labelled r_1 asks for a piece of mobile code (e.g. an applet), and it can be served by two code providers at ℓ_1 and ℓ_2. The request type $\tau \to (\tau \to \tau)$ means that, upon receiving a value of type τ (which can be an arbitrary base type, immaterial here) the invoked service replies with a function from τ to τ, with no security constraints.

The service at ℓ_1 returns a function that protects itself with a policy φ, permitting its use in certified sites only (modelled by the event α_c). Within the function body, the only security-relevant operation is a read α_r on the file system where the delivered code is run. In the public interface of ℓ_1, this behaviour is represented by the history expression $\varphi[\alpha_r]$ which annotates the type of the returned function. The code provided by ℓ_2 first reads (α_r) some local data, and eventually writes them (α_w) back to ℓ_2.

Since ℓ_0 has a limited computational power, the code f obtained by the request r_1 is passed as a parameter to the service invoked by the request r_2. This request can be served by ℓ_3 and ℓ_4. The service at ℓ_3 is certified (α_c), and runs the provided code f under a "Chinese Wall" security policy φ', requiring that no data can be written (α_w) after reading them (α_r). The service at ℓ_4 is not certified, and it simply runs f.

The abstract behaviour of the whole network is rendered by the following history expression H:

$$\{r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}]$$
$$r_2[\ell_4] \rhd \ell_4 : \{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}\}$$

The intuitive meaning of H is that, if r_2 is served by ℓ_3 (written as $r_2[\ell_3]$), then the event α_c is generated at site ℓ_3, followed by a *safety framing* φ'. This framing protects the behaviour it wraps, i.e. $\varphi[\alpha_r]$ if ℓ_1 is chosen for r_1, or $\alpha_r\alpha_w$ if ℓ_2 is chosen instead. Otherwise, if r_2 is served by ℓ_4, then the behaviour (on site ℓ_4) depends on the former choice for r_1: if ℓ_1 was selected, then $\varphi[\alpha_r]$, otherwise $\alpha_r\alpha_w$.

The history expression H approximates the run-time behaviour of each site in the network, and it can be obtained (e.g. through the type and effect system of Sec. 5) as a suitable combination of the abstract behaviour of the client ℓ_0 with the certified interfaces of the services it can invoke.

Our next goal is to determine how to compose services while keeping security, i.e. while respecting all the policies on demand. The composition of services is rendered as a *plan* that chooses the appropriate service for each request. The *viable* plans that drive safe executions are obtained by statically analysing the history expression H inferred for the network. Our analysis first "flattens" the structure of the history expression, by collecting all the possible combinations of service choices. In our example, we would obtain:

$$\begin{aligned} H' = \{&r_1[\ell_1] \mid r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\varphi[\alpha_r]],\\ &r_1[\ell_2] \mid r_2[\ell_4] \rhd \ell_4 : \alpha_r \cdot \alpha_w,\\ &r_1[\ell_1] \mid r_2[\ell_4] \rhd \ell_4 : \varphi[\alpha_r]\\ &r_1[\ell_2] \mid r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w]\} \end{aligned}$$

Every element of H' clearly separates the plan from the associated abstract behaviour, which has no further plans within – and so it can be model-checked for validity using

the techniques in [3]. For instance, under the plan that composes $r_1[\ell_1]$ with $r_2[\ell_3]$ (written as $r_1[\ell_1] \mid r_2[\ell_3]$), the overall abstract behaviour is $\alpha_c \cdot \varphi'[\varphi[\alpha_r]]$. The first two plans in H' are viable, while the others give rise to non-valid behaviour. The plan $r_1[\ell_1] \mid r_2[\ell_4]$ is *not* viable, because the policy φ would be violated when f is run on a non certified site; instead, the plan $r_1[\ell_2] \mid r_2[\ell_3]$ would violate φ'.

Planning service composition can be even more complex. Consider a slight extension of our example, where the client is billed for the services it has invoked. To do that, assume that an argument g is passed to the request r_1, to invoke a billing service through a request r_3, and so let the code provider invoice the customer ℓ_0 for the service. The same function g is also passed later on the service which will actually run the code f, to charge ℓ_0 for the cost of the execution.

A billing service acts as a function that takes as input an invoice (of some type τ') and delivers back a payment certification, i.e. a function of type $\tau' \xrightarrow{\alpha_{paid}} \tau'$ that generates α_{paid} to signal successful transaction. Let $\tau_b = \tau' \to (\tau' \xrightarrow{\alpha_{paid}} \tau')$ be the type of billing services. Then, the request types of r_1 and r_2 would have the following form:

$$\rho_1 = \tau \times \tau_b \xrightarrow{\psi} (\tau \to \tau) \qquad \rho_2 = (\tau \to \tau) \times \tau_b \xrightarrow{\psi} \tau$$

where the property ψ on demand requires that payment is accomplished before the control returns back to the client. The request types ρ_1 and ρ_2 will be exploited to discover services matching both the syntactic signature and the required behaviour ψ.

Assume now that two billing services ℓ_5 and ℓ_6 are discovered in the network. The service ℓ_5 can be used by certified users only, while ℓ_6 imposes no constraints. Clearly, the service which provides the code and the one which runs it can choose different billing services. The plan $r_1[\ell_1.r_3[\ell_6]] \mid r_2[\ell_3.r_3[\ell_5]]$ is viable: the request r_3 is resolved with ℓ_6 within the service ℓ_1 chosen for r_1, while it is resolved with ℓ_5 within the service ℓ_3 chosen for r_2. Instead, the plan $r_1[\ell_1.r_3[\ell_5]] \mid r_2[\ell_3.r_3[\ell_5]]$ is *not* viable, because ℓ_1 is not certified.

3 Programming model

To study secure service orchestration in a formal setting, we consider a calculus where services are functional units distributed over a network. We first define their syntax and *stand-alone* operational semantics, i.e. the behaviour of a service in isolation. We then introduce plans, that drive the selection of services provided by the network. Finally, we define the syntax and operational semantics of networks.

Services. A service is modelled as an expression in a λ-calculus enriched with primitives for security and service requests. Security-relevant operations are rendered as side-effects in the calculus, and they are called *access events*. A *security policy* φ is a regular property over a sequence η of access events, called *history*. The programming construct used to enforce security policies is called *safety framing*: a service e framed within a policy φ (written $\varphi[e]$) must respect φ at each step of its execution. A *service request* has the form $\mathtt{req}_r\rho$. The label r uniquely identifies the request, while the *request type* ρ is the query pattern to be matched by the invoked service. Types are defined afterwards, and are used by the orchestrator to statically discover the plans that drive safe executions. Indeed, our network semantics exploits plans to resolve service requests.

The abstract syntax of services follows. We assume as given the languages for (regular) policies φ and for guards b. We omit their definition here, as they are not relevant for the subsequent technical development. To enhance readability, our calculus comprises conditional expressions and named abstractions (the variable z in $e' = \lambda_z x.\, e$ stands for e' itself within e).

Syntax of services

e, e'	::=	x	variable
		α	access event
		$\mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e$	conditional
		$\lambda_z x.\, e$	abstraction
		$e\, e'$	application
		$\varphi[e]$	safety framing
		$\mathtt{req}_r\tau$	service request
		$\mathtt{wait}\ \ell$	wait reply

The values v of our calculus are the variables, the abstractions, and the requests. We write $*$ for a fixed, closed, event-free value, and $\lambda.\, e$ for $\lambda x.\, e$, for x not free in e. The following abbreviation is standard: $e; e' = (\lambda.\, e')\, e$. Without loss of generality, we assume that framings include at least one event, maybe dummy.

The stand-alone evaluation of a service is much alike the call-by-value semantics of the λ-calculus; additionally, it enforces all the policies within their framings. Since here services are considered in isolation, requests are not resolved. The configurations are pairs η, e. A transition $\eta, e \to \eta', e'$ means that, starting from a history η, the service e evolves to e' and extends η to η'. We write $\eta \models \varphi$ when the history η obeys the policy φ. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Service semantics (stand-alone)

$$\frac{\eta, e_1 \to \eta', e_1'}{\eta, e_1 e_2 \to \eta', e_1' e_2} \qquad \frac{\eta, e_2 \to \eta', e_2'}{\eta, v e_2 \to \eta', v e_2'}$$

$$\eta, (\lambda_z x.\, e) v \to \eta, e\{v/x, \lambda_z x.\, e/z\}$$

$$\eta, \alpha \to \eta\alpha, * \qquad \eta, \texttt{if}\ b\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \to \eta, e_{\mathcal{B}(b)}$$

$$\frac{\eta, e \to \eta', e' \quad \eta' \models \varphi}{\eta, \varphi[e] \to \eta', \varphi[e']} \qquad \frac{\eta \models \varphi}{\eta, \varphi[v] \to \eta, v}$$

The first two rules implement call-by-value evaluation; as usual, functions are not reduced within their bodies. The third rule implements β-reduction. Notice that the whole function body $\lambda_z x.\, e$ replaces the self variable z after the substitution, so giving an explicit copy-rule semantics to recursive functions. The evaluation of an event α consists in appending α to the current history, and producing the no-operation value $*$. A conditional $\texttt{if}\ b\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff}$ evaluates to e_{tt} (resp. e_{ff}) if b evaluates to true (resp. false).

To evaluate a safety framing $\varphi[e]$, we must consider two cases. If, starting from the current history η, e may evolve to e' and extend the history to η', then the whole framing $\varphi[e]$ may evolve to $\varphi[e']$, provided that η' satisfies φ. Otherwise, if e is a value and the current history satisfies φ, then the scope of the framing is left. In both cases, as soon as a history is found not to respect φ, the evaluation gets stuck.

Plans. When a service is plugged into a network, a plan is used to resolve the requests therein, acting as an orchestrator. Our static machinery will deduce plans guaranteeing that the selected services matches the requests. Plans have the following syntax:

Syntax of Plans

π, π'	::=	0	empty
		$r[\ell.\pi]$	service choice
		$\pi \mid \pi'$	composition

The empty plan 0 has no choices. The plan $r[\ell.\pi]$ associates the service e published at site ℓ with the request labelled r, and imposes the plan π to e (so constrained by the choice taken for r). The composition operator $\mid$ is associative, commutative and idempotent. We abbreviate $r[\ell.0]$ with $r[\ell]$. We require plans to have a single choice for each request made at the same level, i.e. $r[\ell.\pi] \mid r[\ell'.\pi']$ implies $\ell = \ell'$ and $\pi = \pi'$.

Networks. A service e is plugged into a network by publishing it at a site ℓ, together with its interface τ. Hereafter, $\ell\langle e : \tau\rangle$ denotes such a *published service*. Labels ℓ can be seen as Uniform Resource Identifiers, and they are only known by the orchestrator. We assume that each site publishes a single service, and that interfaces are certified, i.e. they are inferred by the type system in Sect. 5. Also, we assume that services cannot invoke each other circularly, because this would result in a meaningless service composition. A *client* is a special published service $\ell\langle e : unit\rangle$. As we will see, this special form prevents anyone from invoking a client. A *network* is a set of clients and published services.

The state of a published service $\ell\langle e : \tau\rangle$ is denoted by $\ell\langle e : \tau\rangle : \pi \rhd \eta, e'$, where π is the plan used by the current instantiation of the service, η is the history generated so far, and e' models the code in execution. When unambiguous, we simply write ℓ for $\ell\langle e : \tau\rangle$ in states.

The syntax and the operational semantics of networks follows; the operator $\parallel$ is associative and commutative. Given a network $\{\ell_i\langle e_i : \tau_i\rangle\}_{i\in 1..k}$, a *configuration* N has the form $\ell_1 : \pi_1 \rhd \eta_1, e_1' \parallel \cdots \parallel \ell_k : \pi_k \rhd \eta_k, e_k'$, abbreviated as $\{\ell_i : \pi_i \rhd \eta_i, e_i'\}_{i\in 1..k}$. To trigger a computation of the network, we need to fix the plans π_i for each client; if ℓ_i is a service, we assume $\pi_i = 0$. Then, for all $i \in 1..k$, the initial configuration has $\eta_i = \varepsilon$, and $e_i' = *$ if ℓ_i is a service, while $e_i' = e_i$ if ℓ_i is a client.

Network configurations and semantics

N, N'	::=	$\ell\langle e : \tau\rangle : \pi \rhd \eta, e'$	service state
		$N \parallel N'$	composition

$$\frac{\eta, e \to \eta', e'}{\ell : \pi \rhd \eta, e \to \ell : \pi \rhd \eta', e'} \qquad \frac{N_1 \to N_1'}{N_1 \parallel N_2 \to N_1' \parallel N_2}$$

$$\ell : (r[\ell'.\pi'] \mid \pi'') \rhd \eta, (\texttt{req}_r \rho) v \parallel \ell'\langle e : \tau\rangle : 0 \rhd \varepsilon, * \to \ell : (r[\ell'.\pi'] \mid \pi'') \rhd \eta, \texttt{wait}\ \ell' \parallel \ell'\langle e : \tau\rangle : \pi' \rhd e\, v$$

$$\ell : \pi \rhd \eta, \texttt{wait}\ \ell' \parallel \ell' : \pi' \rhd \eta', v \to \ell : \pi \rhd \eta, v \parallel \ell' : 0 \rhd \varepsilon, *$$

A transition of a stand-alone service is localized at site ℓ, regardless of a plan π. The second rule specifies the asynchronous behaviour of the network: a transition of a subnetwork becomes a transition of the whole network. The last two rules model requests and replies. A request r, resolved by the current plan with the service ℓ', can be served if the service is available, i.e. it is in the state $\ell' : 0 \rhd \varepsilon, *$. In this case, a new instance of the service is generated: e is applied to the received argument v, under the plan π', received as well from the invoker. The special event σ signals that the service has started. The invoker waits until ℓ' has produced a value. When this happens, the service becomes available again. We follow here the *stateless* approach, by

clearing the history of a service at each instantiation (indeed, statefullness could be easily obtained by maintaining the history η' at ℓ' in the last rule).

Note that a service works in a tail-recursive fashion, and so there is a single instance of it in network configurations. We could easily model replication of services, by creating a new instance for each request. Note also that a network evolves by interleaving the activities of its components, which only sinchronize when competing for the same service. It is straightforward to derive a truly concurrent semantics from the above one, e.g. using C/E Petri nets.

4 History expressions

We shall now extend the history expressions of [3]. They statically predict the histories generated at run-time by a network of clients and services.

Syntax. History expressions are much alike context-free specifications, and include the empty history ε, access events α, sequencing $H \cdot H'$, non-deterministic choice $H + H'$, safety framings $\varphi[H]$, recursion $\mu h.H$ (μ binds the occurrences of the variable h in H), localization $\ell : H$, and planned selections $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$.

Syntax of history expressions

$H, H' ::=$	ε	empty
	h	variable
	α	access event
	$H \cdot H'$	sequence
	$H + H'$	choice
	$\varphi[H]$	safety framing
	$\mu h.H$	recursion
	$\ell : H$	localization
	$\{\pi_i \triangleright H_i\}_{i \in I}$	planned selection

Intuitively, access events represent the program actions where sensible resources are accessed; the constructors $\cdot$ and $+$ correspond to sequentialization of code and conditionals, respectively; safety framings model blocks of code subject to security policies; recursion is for loops and recursive functions. These constructs have been previously introduced in [3]. The new construct $\ell : H$ localizes the behaviour H to the site ℓ. For example, $\ell : \alpha \cdot (\ell' : \alpha') \cdot \beta$ denotes two histories: $\alpha\beta$ occurring at location ℓ, and α' occurring at ℓ'. A planned selection abstracts the behaviour of service requests. For instance, $\{r[\ell_1] \triangleright H_1 \cdots r[\ell_k] \triangleright H_k\}$ says that a request r can be resolved into one of the services provided by the sites $\ell_1, \ldots, \ell_k$, which may generate a history represented by $H_1, \ldots, H_k$, respectively.

Semantics. To give a semantics to history expressions, we enrich the set of events with *framing* events of the form $[_\varphi,]_\varphi$, that denote the opening and closing of a safety framing $\varphi[\cdots]$. For example, the history $\eta = \alpha[_\varphi \alpha']_\varphi$ represents a computation that (i) generates an event α, (ii) enters the scope of φ, (iii) generates α' within the scope of φ, and (iv) leaves the scope of φ. Hereafter, we shall only consider histories with balanced framing events.

The *denotational semantics* of a history expression is a set, written $(\ell_i : \mathcal{H}_i)_{i \in I}$. The intended meaning is that the behaviour of the service at location ℓ_i is approximated by the set of histories $\mathcal{H}_i$. Technically, $\mathcal{H}$ belongs to the lifted cpo of sets of histories [27], ordered by (lifted) set inclusion $\subseteq_\perp$ (where $\perp \subseteq_\perp \mathcal{H}$ for all $\mathcal{H}$, and $\mathcal{H} \subseteq_\perp \mathcal{H}'$ whenever $\mathcal{H} \subseteq \mathcal{H}'$). The least upper bound between two elements of the cpo is standard set union $\cup$, assuming that $\perp \cup \mathcal{H} = \mathcal{H}$. For notational convenience, we feel free to omit curly braces when writing singleton sets, and we write $\varphi[\mathcal{H}]$ for $\{\, [_\varphi \eta]_\varphi \mid \eta \in \mathcal{H} \,\}$.

The *stateless* semantics $\langle\!\langle H \rangle\!\rangle^\pi$ of a closed history expression H depends on the given evaluation plan π, and is defined in two steps. In the first, we define the *stateful* semantics $[\![H]\!]^\pi_\theta$ (in an environment θ binding variables), i.e. a semantics in which services keep track of the histories generated by all the past invocations. A simple transformation then yields $\langle\!\langle H \rangle\!\rangle^\pi$, in which each invocation is instead independent of the previous ones, i.e. it always starts with the empty history.

Semantics of history expressions

$$\langle\!\langle H \rangle\!\rangle^\pi = \{\, \ell : \{\, \langle\!\langle \eta \rangle\!\rangle \mid \eta \in \mathcal{H} \,\} \mid \ell : \mathcal{H} \in [\![H]\!]^\pi_\emptyset \,\}$$

$$\text{where } \langle\!\langle \eta \rangle\!\rangle = \begin{cases} \eta & \text{if } \sigma \notin \eta \\ \langle\!\langle \eta_0 \rangle\!\rangle \cup \langle\!\langle \eta_1 \rangle\!\rangle & \text{if } \eta = \eta_0\, \sigma\, \eta_1 \end{cases}$$

$$[\![\varepsilon]\!]^\pi_\theta = (? : \varepsilon) \quad [\![\alpha]\!]^\pi_\theta = (? : \alpha) \quad [\![\ell : H]\!]^\pi_\theta = [\![H]\!]^\pi_\theta\{\ell/?\}$$

$$[\![\varphi[H]]\!]^\pi_\theta = \varphi[[\![H]\!]^\pi_\theta] \qquad [\![H \cdot H']\!]^\pi_\theta = [\![H]\!]^\pi_\theta \odot [\![H']\!]^\pi_\theta$$

$$[\![h]\!]^\pi_\theta = \theta(h) \qquad [\![H + H']\!]^\pi_\theta = [\![H]\!]^\pi_\theta \oplus [\![H']\!]^\pi_\theta$$

$$[\![\mu h.H]\!]^\pi_\theta = \bigoplus_{n \in \omega} f^n(? : \perp) \text{ where } f(X) = [\![H]\!]^\pi_{\theta\{X/h\}}$$

$$[\![\{\pi_i \triangleright H_i\}_{i \in I}]\!]^\pi_\theta = \bigoplus_{\pi_i \sqsubseteq \pi} [\![H_i]\!]^{\pi_i/\pi}_\theta \text{ where } \bigoplus \emptyset = (? : \perp)$$

We first comment on the rules for $[\![H]\!]^\pi_\theta$. The meaning of an event α is the pair $(? : \{\alpha\})$, where ? is dummy and will be bound to the relevant location. The rule for localizing H at ℓ records the actual binding: the current location ℓ replaces "?". The semantics of a sequence $H \cdot H'$ is obtained by con-

catenating the histories denoted by H and H' site by site, using the auxiliary (strict) function $\odot$ defined afterwards. Similarly for the semantics of choices $H + H'$, that joins the histories site by site through the strict operator $\oplus$. The semantics of $\mu h.\, H$ is the least fixed point of the operator f above, computed in the cpo obtained by coalesced sum of the cpos of sets of histories $\mathcal{H}$.

The semantics of a planned selection $\{\pi_i \triangleright H_i\}_{i\in I}$ under an evaluation plan π is the sum of the semantics of those H_i such that π resolves all the choices in π_i (rendered as $\pi_i \sqsubseteq \pi$). In that case, the choices in π_i are consumed, and the evaluation of H_i proceeds with the residual plan π_i/π. If π does not resolve any of the π_i, then no service is available, and an error occurs (rendered as $\bot$). Consider for example the evaluation of $\{r[\ell] \triangleright \ell : \{r'[\ell_1] \triangleright \alpha_1, r'[\ell_2] \triangleright \alpha_2\}\}$ under $\pi = r[\ell.r'[\ell_1]] \mid r'[\ell_2]$. Since π resolves the choice for r, one has to evaluate $\ell : \{r'[\ell_1] \triangleright \alpha_1, r'[\ell_2] \triangleright \alpha_2\}$ under the residual plan $r'[\ell_1]$, obtained by consuming the choice $r[\ell]$. The final result is then $(\ell : \{\alpha_1\})$.

The intuition on the auxilary operators follows. The sequentialization of $(\ell_i : \mathcal{H}_i)_{i\in I}$ and $(\ell'_j : \mathcal{H}'_j)_{j\in J}$ consists of two parts. The first part comprises $\ell_i : \mathcal{H}_i\mathcal{H}_j$ for all $\ell_i = \ell'_j$, i.e. $\ell_i : \{\, \eta\eta' \mid \eta \in \mathcal{H}_i, \eta' \in \mathcal{H}_j \,\}$. The second part has $\ell_i : \mathcal{H}_i$ and $\ell'_j : \mathcal{H}'_j$ for all $i \notin J$ and $j \notin I$. As an example, $(\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1, \beta_1\}) \odot (\ell_1 : \{\gamma_1\}, \ell_2 : \{\alpha_2\}) = (\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1\gamma_1, \beta_1\gamma_1\}, \ell_2 : \{\alpha_2\})$. The choice operator $\oplus$ is pretty the same, except that union replaces language concatenation. For example, $(\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1, \beta_1\}) \oplus (\ell_0 : \bot, \ell_1 : \{\gamma_1\}, \ell_2 : \{\alpha_2\}) = (\ell_0 : \bot, \ell_1 : \{\alpha_1, \beta_1, \gamma_1\}, \ell_2 : \{\alpha_2\})$.

The relation $\sqsubseteq$ is a partial order between plans, whose least element is 0. Intuitively, $r[\ell.\pi_0] \sqsubseteq \pi$ if both plans agree on the service selected for r and on the choices occurring in π_0, i.e. π has the form $r[\ell.\pi_1] \mid \pi'_1$ and $\pi_0 \sqsubseteq \pi_1$. For example, $r_0[\ell_0] \sqsubseteq r_0[\ell_0.r_1[\ell_1]] \mid r_2[\ell_2]$. The definition of the residual plan π'/π relies on the fact that plans ordered by $\sqsubseteq$ form a *meet semi-lattice*, and so the meet $\sqcap$ of any pair of elements always exists. For example, $r_0[\ell_0.0]/r_0[\ell_0.r_1[\ell_1]] = 0/r_1[\ell_1] = r_1[\ell_1]$.

Auxiliary definitions

$$\{\ell_i : \mathcal{H}_i\}_{i\in I} \odot \{\ell'_j : \mathcal{H}_j\}_{j\in J} = \{\ell_i : \mathcal{H}_i\mathcal{H}_j\}_{\ell_i=\ell_j} \cup \{\ell_i : \mathcal{H}_i\}_{i\in(I\cup J)\setminus(I\cap J)}$$

$$\{\ell_i : \mathcal{H}_i\}_{i\in I} \oplus \{\ell'_j : \mathcal{H}_j\}_{j\in J} = \{\ell_i : \mathcal{H}_i \cup \mathcal{H}_j\}_{\ell_i=\ell_j} \cup \{\ell_i : \mathcal{H}_i\}_{i\in(I\cup J)\setminus(I\cap J)}$$

$$0 \sqsubseteq \pi \qquad \pi_0 \mid \pi_1 \sqsubseteq \pi \quad \text{if } \pi_0 \sqsubseteq \pi \text{ and } \pi_1 \sqsubseteq \pi$$

$$r[\ell.\pi_0] \sqsubseteq \pi \quad \text{if } \pi = r[\ell.\pi_1] \mid \pi'_1 \text{ and } \pi_0 \sqsubseteq \pi_1$$

$$0/\pi = \pi \qquad r[\ell.\pi_0]/(r[\ell.\pi_1] \mid \pi'_1) = \pi_0/\pi_1$$

$$(\pi_0 \mid \pi_1)/\pi = (\pi_0/\pi) \sqcap (\pi_1/\pi)$$

Example 1. Consider the history expression:

$$H = \ell_0 : \alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \triangleright \ell_2 : \sigma \cdot \alpha_2\} \cdot \beta_0$$

The stateful semantics of H under plan $\pi = r[\ell_1]$ yields:

$$\begin{aligned}
&[\![\alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \triangleright \ell_2 : \sigma \cdot \alpha_2\} \cdot \beta_0]\!]^\pi\{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot [\![\{r[\ell_1] \triangleright \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \triangleright \ell_2 : \sigma \cdot \alpha_2\}]\!]^\pi \\
&\qquad \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot [\![\ell_1 : \sigma \cdot \alpha_1]\!]^0 \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot (\ell_1 : \sigma \cdot \alpha_1) \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= (? : \{\alpha_0\beta_0\},\, \ell_1 : \{\sigma\alpha_1\})\{\ell_0/?\} \\
&= (\ell_0 : \{\alpha_0\beta_0\},\, \ell_1 : \{\sigma\alpha_1\})
\end{aligned}$$

In this case, the stateless semantics just removes the event σ, i.e. $\langle\!\langle H \rangle\!\rangle^\pi = (\ell_0 : \{\alpha_0\beta_0\},\, \ell_1 : \{\alpha_1\})$. □

Example 2. Consider the history expression:

$$H = \ell_0 : \mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \varphi[\sigma \cdot \alpha_1]\} \cdot h$$

This represents a service ℓ_0 that recursively generates α_0 and raise a request r (which can be served by ℓ_1 only). The stateful semantics of H under $\pi = r[\ell_1]$ is:

$$\begin{aligned}
&[\![\ell_0 : \mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \varphi[\sigma \cdot \alpha_1]\} \cdot h]\!]^\pi \\
&= [\![\mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \varphi[\sigma \cdot \alpha_1]\} \cdot h]\!]^\pi\{\ell_0/?\} \\
&= \big(\textstyle\bigoplus_{n\in\omega} [\![f^n(? : \bot)]\!]^\pi\big)\{\ell_0/?\}
\end{aligned}$$

where $f(X) = [\![\beta_0 + \alpha_0 \cdot \{r[\ell_1] \triangleright \ell_1 : \varphi[\sigma \cdot \alpha_1]\} \cdot h]\!]^\pi_{\{X/h\}}$. The fixed point of f, after the substitution $\{\ell_0/?\}$, is:

$$\begin{aligned}
&(\ell_0 : \{\beta_0, \alpha_0\beta_0, \alpha_0\alpha_0\beta_0, \ldots\}, \\
&\ell_1 : \{\varphi[\sigma\alpha_1], \varphi[\sigma\alpha_1]\varphi[\sigma\alpha_1], \ldots\})
\end{aligned}$$

Instead, the stateless semantics $\langle\!\langle H \rangle\!\rangle^\pi$ is the set:

$$(\ell_0 : \{\beta_0, \alpha_0\beta_0, \alpha_0\alpha_0\beta_0, \ldots\}, \ell_1 : \{\varphi[\alpha_1]\})$$ □

Example 3. Consider the history expression:

$$H = \{r[\ell_0] \rhd \{r'[\ell_1] \rhd \ell_1 : \alpha_1, r'[\ell_2] \rhd \ell_2 : \alpha_2\}\}$$

The semantics of H under $\pi = r[\ell_0.r'[\ell_1]] \mid r'[\ell_2]$ is:

$$\begin{aligned}&[\![\{r[\ell_0] \rhd \{r'[\ell_1] \rhd \ell_1 : \alpha_1, r'[\ell_2] \rhd \ell_2 : \alpha_2\}\}]\!]^\pi \\ &= [\![\{r'[\ell_1] \rhd \ell_1 : \alpha_1, r'[\ell_2] \rhd \ell_2 : \alpha_2\}]\!]^{r'[\ell_1]} \\ &= [\![\{r'[\ell_1] \rhd \ell_1 : \alpha_1\}]\!]^{r'[\ell_1]} \;=\; [\![\ell_1 : \alpha_1]\!]^0 \;=\; (\ell_1 : \alpha_1)\end{aligned}$$

In this case there are no σ, so the stateless and the stateful semantics coincide. □

Validity. We now define when histories are valid, i.e. they arise from viable computations that do not violate any security constraint. For example, consider the history $\eta_0 = \alpha_w\alpha_r\varphi[\alpha_w]$, where φ requires that no write α_w occurs after a read α_r. Then, η_0 is *not* valid according to our intended meaning, because the rightmost α_w occurs within a safety framing enforcing φ, and $\alpha_w\alpha_r\alpha_w$ does not obey φ. To be valid, a history η must obey all the policies within their scopes, determined by the framing events in η.

To give a formal definition of validity, it is convenient to introduce the notion of *safe sets*. For example, the history η_0 above has one safe set $\varphi[\{\alpha_w\alpha_r, \alpha_w\alpha_r\alpha_w\}]$. Intuitively, this means that the scope of the framing $\varphi[\cdots]$ spans over the histories $\alpha_w\alpha_r$ and $\alpha_w\alpha_r\alpha_w$. For each safe set $\varphi[\mathcal{H}]$, validity requires that *all* the histories in $\mathcal{H}$ obey φ.

Some notation is now needed. Let $\eta^\flat$ be the history obtained from η by erasing all the framing events, and let η^∂ be the set of all the prefixes of η, including the empty history ε. For example, if $\eta_0 = \alpha_w\alpha_r\varphi[\alpha_w]$, then $(\eta_0^\flat)^\partial = ((\alpha_w\alpha_r[_\varphi\alpha_w]_\varphi)^\flat)^\partial = (\alpha_w\alpha_r\alpha_w)^\partial = \{\varepsilon, \alpha_w, \alpha_w\alpha_r, \alpha_w\alpha_r\alpha_w\}$. Then, the safe sets $S(\eta)$ and validity of histories and of history expressions are defined as follows:

Safe sets and validity

$$\begin{aligned}&S(\varepsilon) = \emptyset \qquad\qquad S(\eta\,\alpha) = S(\eta)\\ &S(\eta_0\,\varphi[\eta_1]) = S(\eta_0\,\eta_1) \cup \varphi[\eta_0^\flat\,(\eta_1^\flat)^\partial]\end{aligned}$$

A history η is *valid* ($\models \eta$ in symbols) when:

$$\varphi[\mathcal{H}] \in S(\eta) \implies \forall \eta' \in \mathcal{H} \;:\; \eta' \models \varphi$$

A history expression H is π-*valid* when, for all ℓ:

$$\langle\!\langle H \rangle\!\rangle^\pi @\, \ell \neq \bot \text{ and } \eta \in \langle\!\langle H \rangle\!\rangle^\pi @\, \ell \implies \models \eta$$

where $(\ell_i : \mathcal{H}_i)_{i\in I} @\, \ell_j \;=\; \mathcal{H}_j$.

Note that validity of a history expression is parametric with the given evaluation plan π, and it is defined componentwise on its semantics, provided it is not $\bot$.

Example 4. The safe sets of the history expression $H = \varphi[\alpha_0 \cdot \{r[\ell_1] \rhd \alpha_1, r[\ell_2] \rhd \varphi'[\alpha_2]\}] \cdot \alpha_3$, with respect to plans $r[\ell_1]$ and $r[\ell_2]$, are:

$$\begin{aligned}S(\langle\!\langle H \rangle\!\rangle^{r[\ell_1]}) &= S([_\varphi\alpha_0\alpha_1]_\varphi\alpha_3) = \{\ \varphi[\{\varepsilon, \alpha_0, \alpha_0\alpha_1\}]\ \}\\ S(\langle\!\langle H \rangle\!\rangle^{r[\ell_2]}) &= S([_\varphi\alpha_0[_{\varphi'}\alpha_2]_{\varphi'}]_\varphi\alpha_3)\\ &= \{\ \varphi[\{\varepsilon, \alpha_0, \alpha_0\alpha_2\}], \varphi'[\{\alpha_0, \alpha_0\alpha_2\}]\ \}\end{aligned}$$

If φ requires "never α_3" and φ' "never α_2", then H is $r[\ell_1]$-valid, because the histories ε, α_0, and $\alpha_0\alpha_1$ obey φ. Instead, H is not $r[\ell_2]$-valid, as the history $\alpha_0\alpha_2$ in the safe set $\varphi'[\{\alpha_0, \alpha_0\alpha_2\}]$ does not obey φ'. □

Example 5. Consider the history expression of Ex. 2 with plan $\pi = r[\ell_1]$, and assume that the policy φ requires "never α_1 more than once". Then, the stateless semantics $\langle\!\langle H \rangle\!\rangle^\pi$ has exactly one safe set, $\varphi[\{\varepsilon, \alpha_1\}]$, which is valid. Instead, the stateful $[\![H]\!]^\pi$ would have the safe set $\varphi[\{\varepsilon, \alpha_1, \alpha_1\alpha_1, \ldots\}]$, which is *not* valid. □

5 Types and effects

We now introduce a type and effect system for our calculus, building upon [3]. Types and type environments, ranged over by τ and Γ, are mostly standard and are defined in the following table. The history expression H in the functional type $\tau \xrightarrow{H} \tau'$ describes the latent effect associated with an abstraction, i.e. one of the histories represented by H is generated when a value is applied to an abstraction with that type.

Types and Type Environments

$$\begin{aligned}&\tau, \tau' ::= \mathit{unit} \mid \tau \xrightarrow{H} \tau'\\ &\Gamma ::= \emptyset \mid \Gamma; x : \tau \qquad \text{where } x \notin \mathit{dom}(\Gamma)\end{aligned}$$

For notational convenience, we assume that the request type ρ in $\mathtt{req}_r\rho$ is a special type. E.g. we use $\mathit{unit} \xrightarrow{\varphi[\varepsilon]} (\mathit{unit} \xrightarrow{\varphi'[\varepsilon]} \mathit{unit})$ for the request type of a service obeying φ and returning a function subject to the policy φ'. Additionally, we put some restrictions on request types. First, only functional types are allowed: this models services being considered as remote procedures (instead, clients have unit type, so they cannot be invoked). Second, no constraints should be imposed over ρ_0 in a request type $\rho_0 \xrightarrow{\varphi} \rho_1$, i.e. in ρ_0 there are no annotations. This is because the constraints on the selected service should not affect its argument.

A typing judgment $\Gamma, H \vdash e : \tau$ means that the service e evaluates to a value of type τ, and produces a history denoted by the effect H. The auxiliary typing judgment $\Gamma, H \vdash_\ell e : \tau$ is defined as the least relation closed under the rules below, and we write $\Gamma, (\ell : H) \vdash e : \tau$ when the service e at ℓ is typed by $\Gamma, H \vdash_\ell e : \tau$. Typing judgments are similar to those of the simply-typed λ-calculus, and improve on those of [3] (see there for worked-out examples with no requests). The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The actual effect of an abstraction is the empty history expression, while the latent effect is equal to the actual effect of the function body. The rule for abstraction constraints the premise to equate the actual and latent effects, up to associativity, commutativity, idempotency and zero of $+$, associativity and zero of $\cdot$, α-conversion, and elimination of vacuous μ-binders. The next-to-last rule allows for weakening of effects. Note that our type system does not assign any type to `wait` expressions: indeed, waits are only needed in configurations, and not in service code.

We stipulated that the services provided by the network have certified types. Consequently, the typing relation is parametrized by the set W of services $\ell\langle e : \tau\rangle$ such that $\emptyset, \varepsilon \vdash_\ell e : \tau$. We assume W to be fixed, and we write $\vdash_\ell$ instead of $\vdash_{\ell,W}$. To enforce non-circular service composition, we require W to be partially ordered by $\prec$, where $\ell \prec \ell'$ if ℓ can invoke ℓ'; clients are obviously the least elements of $\prec$, and they are not related to each other. Note that the up-wards cone of $\prec$ of a client represents the (partial) knowledge it has of the network.

Typing services

$$\frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, \ell : H \vdash e : \tau} \quad \text{if } e \text{ is published at } \ell$$

$$\Gamma, \varepsilon \vdash_\ell * : \mathit{unit} \qquad \Gamma, \alpha \vdash_\ell \alpha : \mathit{unit} \qquad \Gamma, \varepsilon \vdash_\ell x : \Gamma(x)$$

$$\frac{\Gamma, H \vdash_\ell e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_\ell e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_\ell e\, e' : \tau'}$$

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_\ell e : \tau'}{\Gamma, \varepsilon \vdash_\ell \lambda_z x.\ e : \tau \xrightarrow{H} \tau'} \qquad \frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, \varphi[H] \vdash_\ell \varphi[e] : \tau}$$

$$\frac{\Gamma, H \vdash_\ell e : \tau \quad \Gamma, H \vdash_\ell e' : \tau}{\Gamma, H \vdash_\ell \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau} \qquad \frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, H + H' \vdash_\ell e : \tau}$$

$$\frac{\tau = \Cup\{\, \rho \boxplus_{r[\ell']} \tau' \mid \emptyset, \varepsilon \vdash_{\ell'} e : \tau' \quad \ell \prec \ell'\langle e : \tau'\rangle \quad \rho \approx \tau' \,\}}{\Gamma, \varepsilon \vdash_\ell \mathtt{req}_r \rho : \tau}$$

A service invocation $\mathtt{req}_r \rho$ has an empty actual effect, and a functional type τ, whose latent effect is a planned selection that picks from the network those services known by ℓ and matching the request type ρ.

To give a type to requests, we need some auxiliary technical notation. First we introduce $\approx$, $\boxplus$ and $\Cup$, with the help of a running example. We write $\rho \approx \tau$, and say ρ, τ *compatible*, whenever, omitting the annotations on the arrows, ρ and τ are equal. Formally:

$$\mathit{unit} \approx \mathit{unit}$$
$$(\rho_0 \xrightarrow{\varphi} \rho_1) \approx (\tau_0 \xrightarrow{H} \tau_1) \quad \text{iff } \rho_0 \approx \tau_0 \text{ and } \rho_1 \approx \tau_1$$

Example 6. Let $\rho = (\tau \to \tau) \xrightarrow{\varphi} (\tau \to \tau)$, with $\tau = \mathit{unit}$, be the request type in $\mathtt{req}_r \rho$, and consider two services $\ell_i\langle e_i : \tau_i\rangle$ with $\tau_i = (\tau \xrightarrow{h_i} \tau) \xrightarrow{\alpha_i \cdot h_i} (\tau \xrightarrow{\beta_i} \tau)$, for $i \in 1..2$. We have that $\tau_1 \approx \rho \approx \tau_2$, i.e. both the services are compatible with the request r. □

The operator $\boxplus_{r[\ell]}$ combines a request type ρ and a type τ, when they are compatible. Given a request type $\rho = \rho_0 \xrightarrow{\varphi} \rho_1$ and a type $\tau = \tau_0 \xrightarrow{H} \tau_1$, the result of $\rho \boxplus_{r[\ell]} \tau$ is $\tau_0 \xrightarrow{\{r[\ell] \rhd \ell : \varphi[\sigma \cdot H]\}} (\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1)$, where:

$$\mathit{unit}\, \underline{\boxplus}_{r[\ell]} \mathit{unit} = \mathit{unit}$$
$$(\rho_0 \xrightarrow{\varphi} \rho_1) \underline{\boxplus}_{r[\ell]} (\tau_0 \xrightarrow{H} \tau_1) = (\rho_0 \underline{\boxplus}_{r[\ell]} \tau_0) \xrightarrow{\{\underline{r}[\ell] \rhd \varphi[H]\}} (\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1)$$

Example 6 (cont.). The request type ρ is composed with the service types τ_1 and τ_2 as follows:

$$\hat\tau_1 = (\tau \xrightarrow{h_1} \tau) \xrightarrow{\{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot h_1]\}} (\tau \xrightarrow{\{\underline{r}[\ell_1] \rhd \beta_1\}} \tau)$$
$$\hat\tau_2 = (\tau \xrightarrow{h_2} \tau) \xrightarrow{\{r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot h_2]\}} (\tau \xrightarrow{\{\underline{r}[\ell_2] \rhd \beta_2\}} \tau)$$

where $\hat\tau_1 = \rho \boxplus_{r[\ell_1]} \tau_1$ and $\hat\tau_2 = \rho \boxplus_{r[\ell_2]} \tau_2$. □

The top-level arrow carries a planned selection $\{r[\ell] \rhd \ell : \varphi[\sigma \cdot H]\}$, meaning that, if the service at ℓ is chosen for r, then it generates (at location ℓ, and prefixed by σ) the behaviour H, subject to the policy φ. This top-level choice induces a dependency on the further choices for r recorded in $\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1$. These *dependent choices* are written $\underline{r}[\ell]$, and their effect is not localized. In the example above, the service at ℓ_1 returns a function whose (latent) effect $\{\underline{r}[\ell] \rhd \beta_1\}$ means that β_1 occurs in the location where the function will be actually applied. Dependent choices are only used for technical reasons, to improve the precision of the analysis; formally, they are treated just as standard choices. The actual plans provided by the orchestrator will have no dependent choices.

Note that combining functional types never affects the type of the argument. This reflects the intuition that the type

of the argument to be passed to the selected service cannot be constrained by the request.

Eventually, the operator $\Cup$ unifies the types obtained by combining the request type with the service types. Given two types $\tau = \tau_0 \xrightarrow{H} \tau_1$ and $\tau' = \tau'_0 \xrightarrow{H'} \tau'_1$, the result of $\tau \Cup \tau'$ is $\tau''_0 \xrightarrow{H \cup H'} (\tau_1 \varsigma \underline{\Cup} \tau'_1 \varsigma)$, where ς unifies τ_0 and τ'_0 (i.e. $\tau_0\varsigma = \tau'_0\varsigma = \tau''_0$), and:

$$unit \,\underline{\Cup}\, unit = unit$$
$$(\tau_0 \xrightarrow{H} \tau_1) \,\underline{\Cup}\, (\tau'_0 \xrightarrow{H'} \tau'_1) = (\tau_0 \,\underline{\Cup}\, \tau'_0) \xrightarrow{H \cup H'} (\tau_1 \,\underline{\Cup}\, \tau'_1)$$

Example 6 (cont.). We now unify the combination of the request type ρ with the service types, obtaining:

$$\tau' \;=\; (\tau \xrightarrow{h} \tau) \xrightarrow{\{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot h],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot h]\}} (\tau \xrightarrow{\{\underline{r}[\ell_1] \rhd \beta_1,\ \underline{r}[\ell_2] \rhd \beta_2\}} \tau)$$

where $\varsigma = \{h/h_1, h/h_2\}$ is the selected unifier between $\tau \xrightarrow{h_1} \tau$ and $\tau \xrightarrow{h_2} \tau$. □

The following example further illustrates how requests and services are typed.

Example 7. Consider the request and the services of Example 6, and consider the client $(\mathtt{req}_r\rho)(\lambda.\gamma)*$ at site ℓ_0. Note that applying any service resulting from the request r to the function $\lambda.\gamma$ yields a new function, which we eventually apply to the value $*$. We have the typing derivation in Fig. 1. The stateful semantics $[\![H]\!]^\pi$ under $\pi = r[\ell_1]$ is:

$$\begin{aligned}
&[\![\{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma], r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\}]\!]^\pi \\
&\quad \odot [\![\{\underline{r}[\ell_1] \rhd \beta_1,\ \underline{r}[\ell_2] \rhd \beta_2\}]\!]^\pi \{\ell_0/?\} \\
&= (\ell_1 : \{\varphi[\sigma\alpha_1\gamma]\}) \odot (? : \{\beta_1\})\{\ell_0/?\} \\
&= (\ell_1 : \{\varphi[\sigma\alpha_1\gamma]\},\ \ell_0 : \{\beta_1\})
\end{aligned}$$ □

A plan π is *well-typed* for a service at ℓ, $wt_{@\ell}(\pi)$, when, for each request $\mathtt{req}_r\rho$, the chosen service is compatible with ρ, while respecting the partial order $\prec$:

$$\begin{aligned}
&wt_{@\ell}(0) \qquad\qquad wt_{@\ell}(\pi \mid \pi') \text{ if } wt_{@\ell}(\pi) \wedge wt_{@\ell}(\pi') \\
&wt_{@\ell}(r[\ell'.\pi']) \text{ if } \ell \prec \ell'\langle e : \tau\rangle \;\wedge\; \rho \approx \tau \;\wedge\; wt_{@\ell'}(\pi')
\end{aligned}$$

The next theorem states that our type and effect system correctly over-approximates the actual run-time histories. Consider first a network with a single client e at location ℓ_1, and let its computed effect be H, with $\langle\!\langle H \rangle\!\rangle^\pi = (\ell_1 : \mathcal{H}_1, \ldots, \ell_k : \mathcal{H}_k)$ for a given plan π. For each site ℓ_i, the run-time histories occurring therein are prefixes of the histories in $\mathcal{H}_i$ (without framing events). Now, consider a network with $n < k$ clients at the first n sites, each with its own plan π_j and effect H_j. Since clients cannot invoke each other, we have $\langle\!\langle H_j \rangle\!\rangle^{\pi_j} = (\ell_1 : \emptyset, ..., \ell_j : \mathcal{H}_j, ..., \ell_n : \emptyset, \ell_{n+1} : \mathcal{H}_{n+1,j}, ..., \ell_k : \mathcal{H}_{k,j})$. For each service ℓ_i, the run-time histories at ℓ_i belong to (the prefixes of) one of the $\mathcal{H}_{i,j}$, with $1 \le j \le n$ (see Ex. 8). As usual, precision is lost when reducing the conditional construct to nondeterminism, and when dealing with recursive functions.

Theorem 1. Let $\{\ell_i\langle e_i : \tau_i\rangle\}_{i\in I}$ be a network, let N_0 be its initial configuration with all π_i well-typed, and let $\emptyset, H_i \vdash e_i : \tau_i$. If $N_0 \to^* \{\ell_i : \pi'_i \rhd \eta_i, e'_i\}_{i\in I}$, then:

$$\eta_i \in \begin{cases} (\langle\!\langle H_i \rangle\!\rangle^{\pi_i} @\, \ell_i)^\flat\, \partial & \text{if } \ell_i \text{ is a client} \\ (\sigma\langle\!\langle H_j \rangle\!\rangle^{\pi_j} @\, \ell_i)^\flat\, \partial & \text{if } \ell_i \text{ is a service,} \\ & \text{for some client } \ell_j \end{cases}$$

Example 8. Consider a client $e_0 = \alpha_0; (\mathtt{req}_r\rho)*$ at site ℓ_0, with $\rho = unit \to unit$, and a single service $e_1 = \lambda.\alpha_1; \varphi[\mathtt{if}\ b\ \mathtt{then}\ \alpha_2\ \mathtt{else}\ \alpha_3]$ at site ℓ_1, with φ requiring "never α_3". Assume that the guard b always evaluates to true. Then, under the plan $\pi_0 = r[\ell_1]$, we have the following computation:

$$\begin{aligned}
&\ell_0 : \pi_0 \rhd \varepsilon, e_0 \parallel \ell_1 : 0 \rhd \varepsilon, * \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{req}_r\rho * \parallel \ell_1 : 0 \rhd \varepsilon, * \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\ \ell_1 \parallel \ell_1 : 0 \rhd \sigma, e_1 * \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\ \ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \varphi[\mathtt{if} \cdots] \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\ \ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \varphi[\alpha_2] \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\ \ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1\alpha_2, \varphi[*] \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\ \ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1\alpha_2, * \\
\to\ &\ell_0 : \pi_0 \rhd \alpha_0, * \parallel \ell_1 : 0 \rhd \varepsilon, *
\end{aligned}$$

The history expression H_0 extracted from e_0 is:

$$\ell_0 : \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1 \cdot \varphi[\alpha_2 + \alpha_3]\}$$

Then, $\langle\!\langle H_0 \rangle\!\rangle^{\pi_0} = (\ell_0 : \{\alpha_0\},\ \ell_1 : \{\alpha_1[_\varphi\alpha_2]_\varphi, \alpha_1[_\varphi\alpha_3]_\varphi\})$, and the run-time histories generated at site ℓ_1 are strictly contained in the set $(\sigma\langle\!\langle H_0 \rangle\!\rangle^{\pi_0} @ \ell_1)^{\flat\partial} = \{\sigma\alpha_1[_\varphi\alpha_2]_\varphi, \sigma\alpha_1[_\varphi\alpha_3]_\varphi\}^{\flat\partial} = \{\sigma\alpha_1\alpha_2, \sigma\alpha_1\alpha_3\}^\partial = \{\varepsilon, \sigma, \sigma\alpha_1, \sigma\alpha_1\alpha_2, \sigma\alpha_1\alpha_3\}$. □

We can now state the type safety property. We say that a plan π is *viable* for e at ℓ when the evolution of e within a network, under plan π, does not go wrong at ℓ. A computation *goes wrong* at ℓ when it reaches a configuration whose state at ℓ is stuck. A state $\ell : \pi \rhd \eta, e$ is *not stuck* if either $e = v$, or $e = (\mathtt{req}_r\rho)v$, or $e = \mathtt{wait}\ \ell'$, or $\ell : \pi \rhd \eta, e \to \ell : \pi \rhd \eta', e'$. Note that we do not consider requests and waits to be stuck. To see why, consider e.g. the network configuration $\ell_1 : r[\ell_2] \rhd \eta_1, (\mathtt{req}_r\rho)v \parallel \ell_2 : \pi \rhd \eta_2, e \parallel \ell_3 : r[\ell_2] \rhd \eta_3, \mathtt{wait}\ \ell_2$. The client at ℓ_1 is not stuck, because a fair scheduler will allow it to access the service at ℓ_2, as soon as the client at ℓ_3 has obtained a reply.

Theorem 2 (Type Safety). Let $\{\ell_i\langle e_i : \tau_i\rangle\}_{i\in I}$ be a network, and let $\emptyset, H_i \vdash e_i : \tau_i$ for all $i \in I$. If H_i is π_i-valid for π_i well-typed, then π_i is viable for e_i at ℓ_i.

Figure 1. Typing derivation for Example 7

$$\dfrac{\dfrac{\dfrac{\emptyset, \varepsilon \vdash_{\ell_0} \mathtt{req}_r\rho : \tau' \qquad \emptyset, \varepsilon \vdash_{\ell_0} (\lambda.\gamma) : \tau \xrightarrow{\gamma} \tau}{\emptyset, \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \vdash_{\ell_0} (\mathtt{req}_r\rho)(\lambda.\gamma) : \tau \xrightarrow{\{\underline{r}[\ell_1] \rhd \beta_1,\ \underline{r}[\ell_2] \rhd \beta_2\}} \tau}}{\emptyset, \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \cdot \{\underline{r}[\ell_1] \rhd \beta_1,\ \underline{r}[\ell_2] \rhd \beta_2\} \vdash_{\ell_0} (\mathtt{req}_r\rho)(\lambda.\gamma)* : \tau}}{\emptyset, \ell_0 : \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \cdot \{\underline{r}[\ell_1] \rhd \beta_1,\ \underline{r}[\ell_2] \rhd \beta_2\} \vdash (\mathtt{req}_r\rho)(\lambda.\gamma)* : \tau}$$

Example 9. Consider again the network in Ex. 6, where we fix $e_i = \lambda x.\,(\alpha_i; (x*); (\lambda.\beta_i))$ for $i \in 1..2$. Assume the constraint φ on the request type ρ is true. Consider now the client $e_0 = \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*]$ at ℓ_0, where φ_0 requires "never β_2". Let $\pi = r[\ell_1]$. The history expression H_0 of e_0 (inferred as in Ex. 7) is π-valid. Indeed, $\langle\!\langle H_0 \rangle\!\rangle^{\pi} = (\ell_0 : \{\varphi_0[\beta_1]\},\ \ell_1 : \{\varphi[\alpha_1\gamma]\})$, and both $\varphi_0[\beta_1]$ and $\varphi[\alpha_1\gamma]$ are valid. As predicted by Theorem 2, the plan π is viable for e_0 at ℓ_0:

$$\begin{aligned}
&\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*] \parallel \ell_1 : 0 \rhd \varepsilon, * \\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\ \ell_1)*] \parallel \ell_1 : 0 \rhd \sigma, e_1(\lambda.\gamma) \\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\ \ell_1)*] \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \gamma; (\lambda.\beta_1) \\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\ \ell_1)*] \parallel \ell_1 : 0 \rhd \sigma\alpha_1\gamma, (\lambda.\beta_1) \\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\lambda.\beta_1)*] \parallel \ell_1 : 0 \rhd \varepsilon, * \\
\to\ &\ell_0 : \pi \rhd \beta_1, \varphi_0[*] \parallel \ell_1 : 0 \rhd \varepsilon, *
\end{aligned}$$

Note that we have not displayed the configurations at site ℓ_2, because irrelevant here. Consider now the plan $\pi' = r[\ell_2]$. Then H_0 is not π'-valid, because $\langle\!\langle H_0 \rangle\!\rangle^{\pi'} = (\ell_0 : \{\varphi_0[\beta_2]\},\ \ell_2 : \{\varphi[\alpha_2\gamma]\})$, and the event β_2 violates φ_0. In this case the computation:

$$\begin{aligned}
&\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*] \parallel \ell_2 : 0 \rhd \varepsilon, * \\
\to^*\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[\beta_2] \parallel \ell_2 : 0 \rhd \varepsilon, *
\end{aligned}$$

is correctly aborted, because $\beta_2 \not\models \varphi_0$. □

In the following section, we shall present a verification technique that extracts from a history expression the plans that make it valid.

6 Orchestrating services

Once extracted a history expression H from a client e, we have to analyse H to find if there is any viable plan for the execution of e. This issue is not trivial, because the effect of selecting a given service for a request is not confined to the execution of that service. For instance, the history generated while running a service may later on violate a policy that will become active after the service has returned, as shown in Example 10 below. Since each service selection affects the *whole* execution of a network, we cannot simply devise a viable plan by selecting services that satisfy the constraints imposed by the requests, only.

Example 10. Let $e = (\lambda x.\,(\mathtt{req}_{r_2}\rho_2)x)\,((\mathtt{req}_{r_1}\rho_1)*)$, be a client, $\rho_1 = \tau \to (\tau \to \tau)$ and $\rho_2 = (\tau \to \tau) \xrightarrow{\varphi} \tau$, where $\tau = \mathit{unit}$ and φ requires "never γ after β". Intuitively, the service selected upon the request r_1 returns a function, which is then passed as an argument to the service selected upon r_2. Assume the network comprises exactly the following four services:

$$\begin{aligned}
&\ell_1\langle e_{\ell_1} : \tau \xrightarrow{\alpha} (\tau \xrightarrow{\beta} \tau)\rangle \qquad && \ell_2\langle e_{\ell_2} : (\tau \xrightarrow{h} \tau) \xrightarrow{h\cdot\gamma} \tau\rangle \\
&\ell_1'\langle e_{\ell_1'} : \tau \xrightarrow{\alpha'} (\tau \xrightarrow{\beta'} \tau)\rangle \qquad && \ell_2'\langle e_{\ell_2'} : (\tau \xrightarrow{h} \tau) \xrightarrow{\varphi'[h]} \tau\rangle
\end{aligned}$$

where φ' requires "never β'". Since the request type ρ_1 matches the types of e_{ℓ_1} and $e_{\ell_1'}$, both these services can be selected for the request r_1. Similarly, both e_{ℓ_2} and $e_{\ell_2'}$ can be chosen for r_2. Therefore, we have to consider four possible plans when evaluating the history expression H of e:

$$\begin{aligned}
H = \{&r_1[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha, r_1[\ell_1'] \rhd \ell_1' : \sigma \cdot \alpha'\} \cdot \\
&\{r_2[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \{\underline{r_1}[\ell_1] \rhd \beta, \underline{r_1}[\ell_1'] \rhd \beta'\} \cdot \gamma], \\
&\ r_2[\ell_2'] \rhd \ell_2' : \varphi[\sigma \cdot \varphi'[\{\underline{r_1}[\ell_1] \rhd \beta, \underline{r_1}[\ell_1'] \rhd \beta'\}]]\}
\end{aligned}$$

Consider first H under the plan $\pi_1 = r_1[\ell_1] \mid r_2[\ell_2]$, yielding $\langle\!\langle H \rangle\!\rangle^{\pi_1} = (\ell_0 : \emptyset,\ \ell_1 : \{\alpha\},\ \ell_2 : \{\varphi[\beta\gamma]\})$. Then, H is not π_1-valid, because the policy φ is violated at ℓ_2. Consider now $\pi_2 = r_1[\ell_1'] \mid r_2[\ell_2']$, yielding $\langle\!\langle H \rangle\!\rangle^{\pi_2} = (\ell_0 : \emptyset,\ \ell_1' : \{\alpha'\},\ \ell_2 : \{\varphi[\varphi'[\beta']]\})$. Then, H is not π_2-valid, because the policy φ' is violated. Instead, the remaining two plans, $r_1[\ell_1] \mid r_2[\ell_2']$ and $r_1[\ell_1'] \mid r_2[\ell_2]$ are viable for e. □

As shown above, the tree-shaped structure of planned selections makes it difficult to determine the plans under which a history expression is valid. Things become easier if we "linearize" such a tree structure into a set of history expressions, forming an equivalent planned selection $\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, where no H_i has further planned selections. For instance, the linearization of H in Exam-

ple 10 is:

$$\begin{aligned}&\{r_1[\ell_1] \mid r_2[\ell_2] \rhd \ell_1 : \sigma \cdot \alpha \cdot (\ell_2 : \varphi[\sigma \cdot \beta \cdot \gamma]),\\ &\ r_1[\ell_1] \mid r_2[\ell'_2] \rhd \ell_1 : \sigma \cdot \alpha \cdot (\ell'_2 : \varphi[\sigma \cdot \varphi'[\beta]]),\\ &\ r_1[\ell'_1] \mid r_2[\ell_2] \rhd \ell'_1 : \sigma \cdot \alpha' \cdot (\ell_2 : \varphi[\sigma \cdot \beta' \cdot \gamma]),\\ &\ r_1[\ell'_1] \mid r_2[\ell'_2] \rhd \ell'_1 : \sigma \cdot \alpha' \cdot (\ell'_2 : \varphi[\sigma \cdot \varphi'[\beta']])\}\end{aligned}$$

We say that H is *equivalent* to H' ($H \equiv H'$ in symbols) when $\langle\!\langle H \rangle\!\rangle^\pi = \langle\!\langle H' \rangle\!\rangle^\pi$, for each plan π. Also, a history expression H is *linear* when $H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, the plans are pairwise *independent* (i.e. $\pi_i \not\sqsubseteq \pi_j$ for all $i \neq j$) and no H_i has planned selections. The following properties of $\equiv$ hold.

Equational properties of planned selections

$$H \equiv \{0 \rhd H\} \tag{1}$$

$$\{\pi_i \rhd H_i\}_{i\in I} \cdot \{\pi'_j \rhd H'_j\}_{j\in J} \equiv \{\pi_i \mid \pi'_j \rhd H_i \cdot H'_j\}_{i\in I, j\in J} \tag{2}$$

$$\{\pi_i \rhd H_i\}_{i\in I} + \{\pi'_j \rhd H'_j\}_{j\in J} \equiv \{\pi_i \mid \pi'_j \rhd H_i + H'_j\}_{i\in I, j\in J} \tag{3}$$

$$\varphi[\{\pi_i \rhd H_i\}_{i\in I}] \equiv \{\pi_i \rhd \varphi[H_i]\}_{i\in I} \tag{4}$$

$$\mu h.\, \{\pi_i \rhd H_i\}_{i\in I} \equiv \{\pi_i \rhd \mu h.\, H_i\}_{i\in I} \tag{5}$$

$$\{\pi_i \rhd \{\pi'_{i,j} \rhd H_{i,j}\}_{j\in J}\}_{i\in I} \equiv \{\pi_i \boxdot \pi'_{i,j} \rhd H_{i,j}\}_{i\in I, j\in J} \tag{6}$$

where both operands of (2) and (3) are in linear form, and where
$0 \boxdot \pi = \pi \qquad r[\ell.\pi'] \boxdot \pi = r[\ell.\pi' \boxdot \pi]$
$(\pi_0 \mid \pi_1) \boxdot \pi = (\pi_0 \boxdot \pi) \mid (\pi_1 \boxdot \pi)$

The side condition in equations (2) and (3) is easily fulfilled: it suffices to apply the (oriented) equations with the leftmost-innermost evaluation rule. The following theorem enables us to put history expressions in linear form, preserving their semantics.

Theorem 3. The relation $\equiv$ is a congruence, and it satisfies the equations displayed in the table above.

Example 11. Let $H = \mu h.\, \{r[\ell_1] \rhd \alpha_1,\ r[\ell_2] \rhd \alpha_2\} \cdot h$. Then, using equations (1), (6) and (2), we obtain:

$$\begin{aligned}H &\equiv \mu h.\, \{r[\ell_1] \rhd \{0 \rhd \alpha_1\},\ r[\ell_2] \rhd \{0 \rhd \alpha_2\}\} \cdot \{0 \rhd h\}\\ &\equiv \mu h.\, \{r[\ell_1] \boxdot 0 \rhd \alpha_1,\ r[\ell_2] \boxdot 0 \rhd \alpha_2\} \cdot \{0 \rhd h\}\\ &= \mu h.\, \{r[\ell_1] \rhd \alpha_1,\ r[\ell_2] \rhd \alpha_2\} \cdot \{0 \rhd h\}\\ &\equiv \mu h.\, \{r[\ell_1] \mid 0 \rhd \alpha_1 \cdot h,\ r[\ell_2] \mid 0 \rhd \alpha_2 \cdot h\}\\ &\equiv \mu h.\, \{r[\ell_1] \rhd \alpha_1 \cdot h,\ r[\ell_2] \rhd \alpha_2 \cdot h\}\\ &\equiv \{r[\ell_1] \rhd \mu h.\, \alpha_1 \cdot h,\ r[\ell_2] \rhd \mu h.\, \alpha_2 \cdot h\}\end{aligned}$$

Note that the original H can choose a service among ℓ_1 and ℓ_2 at *each* iteration of the loop. Instead, in the linearization of H, the request r will be resolved into the *same* service at each iteration. A slight extension of our machinery can manage plans that are allowed to choose among a set of services for each request. Here, the linearization of H would comprise also the component $r[\{\ell_1, \ell_2\}] \rhd \mu h.\, (\alpha_1 + \alpha_2) \cdot h$, see [4] for details. □

Example 12. Consider the history expression:

$$\begin{aligned}H \ = \ &\varphi[\{r[\ell_1] \rhd \{r_1[\ell_2] \rhd \alpha, r_1[\ell_3] \rhd \beta\}\} \cdot\\ &\{r'[\ell_1] \rhd \{r_1[\ell_2] \rhd \alpha, r_1[\ell_3] \rhd \beta\}\}]\end{aligned}$$

The linearization of H is obtained as follows:

$$\begin{aligned}H &\equiv \varphi[\{r[\ell_1] \boxdot r_1[\ell_2] \rhd \alpha, r[\ell_1] \boxdot r_1[\ell_3] \rhd \beta\} \cdot\\ &\qquad \{r'[\ell_1] \boxdot r_1[\ell_2] \rhd \alpha, r'[\ell_1] \boxdot r_1[\ell_3] \rhd \beta\}]\\ &\equiv \varphi[\{r[\ell_1.r_1[\ell_2]] \rhd \alpha, r[\ell_1.r_1[\ell_3]] \rhd \beta\} \cdot\\ &\qquad \{r'[\ell_1.r_1[\ell_2]] \rhd \alpha, r'[\ell_1.r_1[\ell_3]] \rhd \beta\}]\\ &\equiv \{r[\ell_1.r_1[\ell_2]] \mid r'[\ell_1.r_1[\ell_2]] \rhd \varphi[\alpha \cdot \alpha],\\ &\qquad r[\ell_1.r_1[\ell_2]] \mid r'[\ell_1.r_1[\ell_3]] \rhd \varphi[\alpha \cdot \beta]\\ &\qquad r[\ell_1.r_1[\ell_3]] \mid r'[\ell_1.r_1[\ell_2]] \rhd \varphi[\beta \cdot \alpha],\\ &\qquad r[\ell_1.r_1[\ell_3]] \mid r'[\ell_1.r_1[\ell_3]] \rhd \varphi[\beta \cdot \beta]\}\end{aligned}$$

If φ asks "never $\alpha\alpha$ nor $\beta\beta$", then both $\varphi[\alpha \cdot \beta]$ and $\varphi[\beta \cdot \alpha]$ are valid, and so the corresponding plans $r[\ell_1.r_1[\ell_2]] \mid r'[\ell_1.r_1[\ell_3]]$ and $r[\ell_1.r_1[\ell_3]] \mid r'[\ell_1.r_1[\ell_2]]$ are viable for H. □

Given a history expression H, we obtain its linearization in three steps. First, we apply equation (1) to each event, variable and ε in H. Then, we orient the equations of Theorem 3 from left to right, obtaining a rewriting system that is easily proved finitely terminating and confluent – up to the equational laws of the algebra of plans. The resulting planned selection $H' = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$ has no further selections in H_i, but there may be non-independent plans $\pi_i \sqsubseteq \pi_j$. In the third linearization step, for each such pairs, we update H' by inserting $\pi_i \rhd H_i + H_j$, and removing $\pi_j \rhd H_j$. Note that linearization can produce components $\pi \rhd H$ where π is ill-formed, in the sense that it maps a request to different services, within the same context. For instance, $r_0[\ell_0.r_1[\ell_1]] \mid r_0[\ell_0.r_1[\ell_2]]$ is ill-formed, while $r_0[\ell_0.r_1[\ell_1]] \mid r_2[\ell_0.r_1[\ell_2]]$ is not. We always dispose such ill-formed components.

The following result enables us to detect the viable plans for service composition: executions driven by any of them will never violate security.

Theorem 4. If $H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$ is linear, and H_i is 0-valid for some $i \in 1..k$, then H is π_i-valid.

Summing up, we extract from an client e a history expression H, we linearize it into $\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, and if some H_i is valid, then we can deduce that H is π_i-valid. By Theorem 2, the plan π_i safely drives the execution of e,

without resorting to any run-time monitor. To verify the validity of history expressions that have no planned selections, it suffices to apply the model-checking technique of [3].

7 Conclusions and related work

A static approach has been proposed to study secure orchestration of services. We have presented a distributed calculus with primitives for invoking services that respect given security requirements. The actual histories that can occur at runtime are over-approximated by a type and effect system. These approximations are model-checked to find the plans that guarantee secure executions, without the need of execution monitoring.

We have extended our previous work [3] in two important directions. First, we have modelled more faithfully networks of clients and services, by introducing an explicit notion of location and of located executions. Distribution has offered the advantage of running several clients and services concurrently, but it has made complex to determine and select the behaviour of the single components of the network. Second, we have devised a way of statically constructing the plans that drive succesful, secure executions. In [3], all the services matching the property imposed by a request contributed to the validity of the history expression. Thus, even a single service not respecting the global security requirements sufficed to invalidate the overall history expression, which could no longer be used to devise viable compositions. Instead, here we consider all the plans for service composition one-by-one, and we single out those guaranteeing secure executions.

A possible direction for future research is to extend the applicability of our method, to deal with networks where services can be discovered and deleted on-the-fly. Multi-choice plans [4] are a first solution to deal with services that become unavailable, because they offer many choices for the same request. Publication of new services poses instead a major problem. To cope with that, one has to reconfigure plans at run-time, by exploiting the new interfaces. However, incrementally checking viability of plans is an open problem. A possible solution is to enrich history expressions with *hooks* where new services can be attached. The orchestrator then needs to check the validity of the newly discovered plans, hopefully in an incremental manner.

Related Work. The secure composition of components underlies Sewell and Vitek's box-π [23], an extension of the π-calculus that can express safety policies in the form of *security wrappers*. These are programs that encapsulate a component to control the interactions with other (possibly untrusted) components. A type system that statically captures the allowed causal information flows between components. Our safety framings are closely related to wrappers.

Hennessy, Rathke and Yoshida [14] propose a language for distributed systems, called SAFEDPI. This language allows for processes which may migrate between sites in a controlled manner. The protection model relies on capability-based types. The available resources and their usage policies are modelled respectively as channels and capability types. Process immigration is controlled by (typed) *ports* on the host location: roughly, the type of a port constraints the resources an incoming process can use. Gorla, Hennessy and Sassone [13] consider a similar calculus, where each site has a *membrane* that represents both a security policy and a classification of the levels of trust of external sites. A membrane guards the incoming agents before allowing them to execute.

Recently, increasing attention has been devoted to express service contracts as behavioural (or session) types. These synthetise the essential aspects of the interaction behaviour of services, while allowing efficient static verification of properties of composed systems. Session types [15] have been exploited to formalize compatibility of components [26] and to describe adaptation of web services [8]. Security issues have been recently considered in terms of session types, e.g. in [7], which proves the decidability of type-checking in an extension of the π-calculus with session types and correspondence assertions [28].

Other works have proposed type-based methodologies to check security properties of distributed systems. For instance, Gordon and Jeffrey [12] use a type and effect system to prove authenticity properties of security protocols. Web service authentication has been recently modelled and analysed in [5, 6] through a process calculus enriched with cryptographic primitives.

The problem of discovering and composing Web Services by taking advantage of semantic information has been the subject of a considerable amount of research and development, [2, 9, 17, 19, 22, 25] to cite a few. The idea is to extend the primitives of service description languages with basic constructs for specifying properties of the published interface. We can distinguish between semantic-web descriptions [2, 19, 22, 25] in which service interfaces are annotated with parameter ontologies, and behavioural description [9, 17] in which the annotation details the ordering of service actions. A different solution to planning service composition has been proposed in [16], where the problem of composing services in order to achieve a given goal is expressed as a constraint satisfaction problem. Our approach extends and complements those based on behavioral descriptions, with an eye to security. Indeed, our methodology fully automates the process of discovering services and planning their composition in a secure way.

Acknowledgments. We thank the anonymous referees for their insightful comments. Research partially supported by the EU, within the FETPI Global Computing, Project IST-

2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. of 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] R. Akkiraju et al. *Web Service Semantics*. WSDL-S technical note (version 1.0), 2005.

[3] M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *Proc. of 18th Computer Security Foundations Workshop (CSFW)*, 2005.

[4] M. Bartoletti, P. Degano, and G. L. Ferrari. Plans for service composition. In *Workshop on Issues in the Theory of Security*, 2006.

[5] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *Proc. of ACM Workshop on Secure Web Services*, 2004.

[6] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *Proc. of ACM Symposium on Principles of Programming Languages*, 2004.

[7] E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *Proc. of Foundations of Global Ubiquitous Computing*, 2004.

[8] A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. In *Proc. of AMAST*, 2004.

[9] A. Brogi and R. Popescu. Towards semi-automated workflow-based aggregation of web services. In *Proc. of ICSOC*, 2005.

[10] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarane. The next step in web services. *Communications of the ACM*, 46(10), 2003.

[11] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, 1986.

[12] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. of IEEE Computer Security Foundations Workshop*, 2002.

[13] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In *Proc. of FGUC*, 2004.

[14] M. Hennessy, J. Rathke, and N. Yoshida. SAFEDPI: a language for controlling mobile code. In *Proc. of Fossacs*, 2004.

[15] K. Honda, V. Vansconcelos, and M. Kubo. Language primitives and type discipline for structures communication-based programming. In *Proc. of ESOP*, 1998.

[16] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Constraint Programming CP*, 2005.

[17] S. B. Mokhtar, N. Georgantas, and V. Issarny. Ad hoc composition of user tasks in pervasive computing environment. In *Software Composition*, 2005.

[18] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.

[19] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matchmaking of web services capabilities. In *First International Semantic Web Conference on The Semantic Web*, 2002.

[20] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, 2003.

[21] M. Papazouglou and D. Georgakopoulos. Special issue on service oriented computing. *Communications of the ACM*, 46(10), 2003.

[22] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth. Enhancing web services description and discovery to facilitate composition. In *Semantic Web Services and Web Process Composition*, 2005.

[23] P. Sewell and J. Vitek. Secure composition of untrusted code: box-π, wrappers and causality types. *Journal of Computer Security*, 11(2), 2003.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 2(111), 1994.

[25] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. of ISWC*, 2004.

[26] A. Vallecillo, V. Vansconcelos, and A. Ravara. Typing the behaviours of objects and components using session types. In *Proc. of FOCLASA*, 2002.

[27] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[28] T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, 1993.