# Types and Effects for Resource Usage Analysis

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, Roberto Zunino

Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangi, zunino}@di.unipi.it

**Abstract.** An extension of the $\lambda$-calculus is proposed, to study resource usage analysis and verification. Resources can be dynamically created, and passed / returned by functions; their usages have side effects, represented by events. Usage policies are properties over histories of events, and have a possibly nested, local scope. A type and effect system over-approximates the set of histories a program can generate at run-time. A crucial point solved here concerns correctly associating fresh resources with their usages within approximations. A second issue is that these approximations may contain an unbounded number of fresh resources. Despite of that, we have devised a technique to model-check validity of approximations. A program with a valid approximation is resource-safe: no run-time monitor is needed to safely drive its executions.

## 1 Introduction

An important aspect of programming language design and implementation is how to ensure that resources are used correctly. The typical run-time mechanisms for enforcing resource usage policies are *execution monitors*, which abort executions whenever about to violate the usage policy prescribed by the programmer. The events observed by these monitors are accesses to sensible resources, e.g. opening socket connections, reading/writing files, allocating/deallocating memory. A main issue is finding a compromise between the expressivity of usage policies and the efficiency of the enforcement mechanism. Static analysis techniques may be applied to improve efficiency, but this often results in an unacceptable restriction of the expressive power of policies.

A common mechanism for enforcing usage policies consists in guarding with *local checks* the program points where critical resources can be accessed [10, 18]. Local checks have a main drawback: they must be explicitly inserted into code by the programmer. Since forgetting even a single check might compromise the safety of the whole application, programmers have to inspect very carefully their code. This may be cumbersome even for small programs, and it may easily lead to unnecessary checking.

A safer approach is that of *global policies*, where the execution monitor enforces a global invariant that must hold at any point of the execution. This may involve guarding each resource access, and ad-hoc optimizations are then in order to recover efficiency, e.g. compiling the global policy to local checks [7, 14]. Furthermore, a large monolithic policy may be hard to understand, and not very flexible either. Indeed, one has to imagine all the possible resource usage scenarios in advance. If an unexpected situation occurs at run-time (e.g. a piece of mobile code with specific resource usage requirements), the global policy must be dynamically updated, if possible at all.

A more flexible approach consists in attaching usage policies to resources, so to adapt them to the context where a resource is used. For example, one may restrain the capabilities before calling untrusted code. In [12], a type system extracts from programs an approximation of their possible run-time usage behaviour. Usage policies are arbitrary sets of permitted histories, so statically verifying whether the permitted usages include the extracted approximation is undecidable. Run-time monitoring is thus still needed, unless one restricts to some decidable fragments. As for expressiveness, a limitation is that you can only control the usage of resources you have created. In a mobile code scenario, e.g. a browser that runs untrusted applets, it is also important that you can impose constraints on how external programs manage the resources created in your local environment. For example, an applet may create an unbounded number of resources on the browser site, and never release them. This clearly leads to denial-of-service attacks, that may eventually crash the whole system.

We consider here a language that aims at reconciling expressivity of resource usage policies with efficiency of the enforcement mechanism. This language, called $\lambda^{[\,]}$ (lambda-box), has primitives for creating and accessing resources, and for defining *local* resource usage policies. Sequences of resource accesses in executions are called *histories*; a *policy* is a regular property of histories. A program fragment $e$ protected by a policy $\varphi$ is written $\varphi[e]$, called *policy framing*. Roughly, while evaluating $e$, the histories must respect the policy $\varphi$. Of course, framings can be nested.

Local policies generalise both local checks and global policies. They smoothly allow for safe composition of programs with their own private policies, also in mobile code scenarios. Indeed, there is no need to dynamically accommodate the local private policies into a single global one, possibly invalidating syntax-directed optimizations of the enforcement mechanism. Local policies may offer protection also in the web-services scenario [3]: there, one has not full control on the code to run, and thus inserting local checks is infeasible. For example, a browser must obey a usage policy specified by the user. Additionally, the browser can invoke a policy provider to obtain a stricter security policy, used for dynamically sandboxing applets. This rich interplay between policies seems difficult to express in the above-mentioned approaches.

In $\lambda^{[\,]}$, efficiency of resource usage control is obtained through a suitable combination of static techniques. The type and effect system over-approximates the run-time usage behaviour of a program, by inferring a *history expression* that denotes all the possible histories resulting from executions. A history expression is *valid* when it contains permitted usage patterns only; a program with a valid history expression will never go wrong. Validity of history expressions is then verified through model-checking.

This approach was originally introduced in [1] to deal with *history-based access control*. The present version extends [1] with dynamic creation of resources. This apparently little extension demands for addressing a more general problem, from various viewpoints: one has to correctly bind the creation of new resources to their usages. The solution to this problem deeply affects the techniques of [1], with respect to the following points: (i) the enforcement mechanism, (ii) the semantics of history expressions, (iii) the type and effect system, and (iv) the verification technique.

For the first point, we introduce *template usage automata*: they are an extension of finite state automata (FSA) where the input alphabet is parametrized over resources. A

policy $\varphi$ is represented by a template usage automaton $A_{\varphi(x)}$. To enforce $\varphi$, the usage histories of each resource $r$ must be accepted by the FSA $A_{\varphi(r)}$, obtained by instantiating $A_{\varphi(x)}$ on $r$. For (ii), the semantics of a history expression is a set of histories: the problem here is to equate those histories that only differ in the name of fresh resources. For (iii), the problem is to correctly record the binding of fresh names in history expressions. Constructing the history expression of a program is a basic step in our approach: indeed, checking that a program obeys the usage policies requires knowing all its possible histories in their entirety — history safety is *not* compositional. Technically, we explore a novel approach to quantify types over freshly created resources — a sort of polymorphism à la ML on *both* types and effects. We avoid using explicit binders in types: the definition/use of resources is determined after the type & effect has been inferred. Living without binders made the type and effect system simpler (and required some little ingenuities in proofs). For (iv), the creation of new resources may give rise to an infinite number of formulae to be inspected while verifying validity. We solve this problem by suitably grouping resources with equivalent usage constraints. This allows us to extract from a history expression a Basic Process Algebra [5] and a regular formula, to be used in model-checking validity [9].

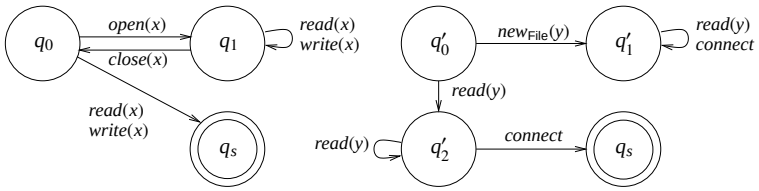
A key point of our proposal is that we offer a comprehensive framework for safely handling resources, within a linguistic setting. On the one hand, our calculus has an expressive and flexible way to compose and enforce usage policies. On the other hand, resource usage control is made feasible by suitably extending and integrating techniques from type theory and model-checking.

## 2 Programming model

We consider a call-by-value $\lambda$-calculus enriched with primitives for creating and accessing resources, and with local usage policies. Resources $r, r', \ldots \in \mathsf{Res}$ are system objects that can be either statically available in $\mathsf{Res}_0 \subset \mathsf{Res}$ or created dynamically. We assume that resources can be accessed through a given finite set of actions $\alpha, \alpha', \ldots \in \mathsf{Act}$. This set is partitioned to reflect the kinds of resources, i.e. $\mathsf{Act} = \bigcup_i \mathsf{Act}_i = \mathsf{File} \cup \mathsf{Socket} \cup \cdots$, where each element of the partition contains the actions admissible for the given kind (e.g. $\mathsf{File} = \{new_{\mathsf{File}}, open, close, read, write\}$). The action $new_{\mathsf{Act}_i}$ represents the creation of a new resource of kind $\mathsf{Act}_i$. An *event* $\alpha(r)$ denotes accessing the resource $r$ through the action $\alpha$. We assume a global *capability environment* $\Gamma_0$ that maps each resource in $\mathsf{Res}_0$ to the set of actions it admits. A *history* $\eta$ is a sequence of access events.

***Usage automata.*** Usage policies $\varphi, \varphi', \ldots \in \mathsf{Pol}$ are regular properties of histories. Each of them is modelled by a *template usage automaton* $A_{\varphi(x)} = \langle Q, q_0, q_s, E \rangle$, which gives rise to a FSA when the parameter $x$ is instantiated to an actual resource $r$. As usual, $Q$ is a *finite* set of states, $q_0 \in Q$ is the start state, while $q_s \in Q$ is the final *sink* state, and $E$ is a finite set of *template edges* of the form $q \xrightarrow{\vartheta} q'$, where $\vartheta \in \{\, \alpha(x), \alpha(\bar{x}), \alpha(r) \mid \alpha \in \mathsf{Act} \wedge r \in \mathsf{Res}_0 \,\}$. The wildcard $\bar{x}$ stands for "any resource different from $x$".

*Example 1.* Consider a file usage policy $\varphi$ saying that only open files can be read or written, and a security policy $\varphi'$ saying that, after having read a file you have not created, you can no longer connect to the network (a sort of "Chinese Wall" property). These policies are described below by the template automata $A_{\varphi(x)}$ (left) and $A_{\varphi'(y)}$ (right, where the resource associated with the action *connect* is irrelevant).




A template usage automaton is *well-kinded* when the resources $r$ in its edges are accessed according to the capability environment $\Gamma_0$, i.e. an edge labelled $\alpha(r)$ requires that $\alpha \in \Gamma_0(r)$. Also, the parameter $x$ must be used consistently: for example, if $x$ is used as a file, e.g. in *read*$(x)$, then it cannot be used also as a socket, or as a printer. To this purpose, we define the kinding function:

$$\kappa(\varphi) = \kappa(\{\, \alpha \mid \exists \zeta \in \{x, \bar{x}\} : q \xrightarrow{\alpha(\zeta)} q' \in E_{\varphi(x)} \,\}) \quad \kappa(\gamma) = \begin{cases} \mathsf{Act}_i & \text{if } \exists i.\, \emptyset \subset \gamma \subseteq \mathsf{Act}_i \\ \emptyset & \text{otherwise} \end{cases}$$

and we require that $\kappa(\varphi) \neq \emptyset$ for all $\varphi$. We assume our automata be always well-kinded.

Given a finite set of resources $R$, a template usage automaton $A_{\varphi(x)}$ is instantiated into a FSA $A_{\varphi(r,R)}$ by binding $x$ to the resource $r \in R$ (we simply write $A_{\varphi(r)}$ when unambiguous). Intuitively, a template edge $q \xrightarrow{\alpha(x)} q'$ results in a transition $\langle q, \alpha(r), q' \rangle$. The instantiation $A_{\varphi(r,R)}$ is $\langle Q, q_0, \Sigma, \delta, F \rangle$, where $\Sigma = \{\, \alpha(r') \mid \alpha \in \mathsf{Act} \wedge r' \in R \,\}$, $F = \{q_s\}$, and the transition relation $\delta : Q \times \Sigma \times Q$ is defined as follows:

$$\bar{\delta} = \{\langle q, \alpha(r), q' \rangle \mid q \xrightarrow{\alpha(x)} q' \,\} \cup \{\langle q, \alpha(r'), q' \rangle \mid q \xrightarrow{\alpha(r')} q' \} \cup \bigcup_{r' \in R \setminus \{r\}} \{\langle q, \alpha(r'), q' \rangle \mid q \xrightarrow{\alpha(\bar{x})} q' \}$$

$$\delta = \bar{\delta} \cup \{\, \langle q, \alpha(r'), q \rangle \mid r' \in R, \nexists \langle q, \alpha(r'), q' \rangle \in \bar{\delta} \,\} \cup \{\, \langle q, \alpha(?), q' \rangle \mid \exists \zeta : q \xrightarrow{\alpha(\zeta)} q' \,\}$$

In the first line we instantiate $x$ to the given resource $r$, we maintain the transitions $\alpha(r')$ for $r' \in \mathsf{Res}_0$, and we instantiate $\alpha(\bar{x})$ with all $r' \neq r$. In the second line we add self-loops for all the events not explicitly mentioned in the template automaton. The last set is only used in the verification phase; the meaning of the special symbol ? will be explained later. Note that finiteness of $R$ and of $\mathsf{Act}$ guarantees that $A_{\varphi(r)}$ is always a finite state automaton. The assumption that $R$ is finite causes no loss of generality, because each time a template usage automaton is instantiated in $\lambda^{[]}$ executions, the number of resources occurring in the history is finite. We denote with $\mathcal{L}(\varphi(r, R))$ the language *not* accepted by $A_{\varphi(r)}$ — thus going into the sink state represents a violation of the policy. Also, $\ell(\varphi)$ stands for the set of actions and resources labelling the template edges.

***The language*** $\lambda^{[]}$**.** The syntax of $\lambda^{[]}$ comprises variables $x, y, \ldots \in \mathsf{Var}$, resources $r, r', \ldots \in \mathsf{Res}$, events $\alpha(e)$, abstractions $\lambda_z x.\, e$ (where $z$ within $e$ stands for the whole abstraction), applications $e\, e'$, conditional expressions `if` $b$ `then` $e$ `else` $e'$ (the definition

of guard $b$ is irrelevant here), policy framings $\varphi[e]$, and resource creation $\texttt{new}\ x : \gamma\ \texttt{in}\ e$, where $\gamma \subseteq \mathsf{Act}$ is the set of capabilities associated with the new resource.

Variables, resources, abstractions, and *failures* are the values $v, v', \ldots$ of $\lambda^{[]}$. A failure $\mathit{fail}_\kappa$ occurs when a computation is about to access a resource of the wrong kind. A failure $\mathit{fail}_{\varphi(r)}$ is raised when about to violate the policy $\varphi(r)$. Let $\mathit{fail}$ denote both kinds of failures. We assume that, for any expression $e$, policy $\varphi$, and action $\alpha$: $\mathit{fail}\ = e\,\mathit{fail}\ = \mathit{fail}\ e\ = \varphi[\mathit{fail}\,] = \alpha(\mathit{fail}\,)$. We write $*$ for a fixed, closed, access-free, non-failure value, and $\lambda.\, e$ for $\lambda x.\, e$ when $x \notin \mathit{fv}(e)$ ($x$ not free in $e$). The following abbreviations are standard: $e; e' = (\lambda.\, e')\, e$, and $\texttt{let}\ x = e\ \texttt{in}\ e' = (\lambda x.\, e')\, e$. We write $\alpha$ instead of $\alpha(r)$ when the parameter $r$ is immaterial. W.l.o.g. we assume that each framing has an opening event, i.e. for all $\varphi[e]$, the expression $e$ has the form $\alpha; e'$, for some $\alpha$ and $e'$. The opening event can be dummy, with no influence on usage policies.

We define the behaviour of $\lambda^{[]}$ expressions through the following small-step operational semantics. A transition $\eta, \Gamma, e \to \eta', \Gamma', e'$ means that, starting from a history $\eta$ and capability environment $\Gamma$, the expression $e$ may evolve to $e'$ (possibly a failure), the history $\eta$ to $\eta'$ and the capability environment $\Gamma$ to $\Gamma'$. Initial configurations have the form $\varepsilon, \Gamma_0, e$, where $\varepsilon$ denotes the empty history.

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

$$\eta, \Gamma, (\lambda_z x.e)v \to \eta, \Gamma, e\{v/x, \lambda_z x.e/z\} \qquad \eta, \Gamma, \texttt{if}\ b\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \to \eta, \Gamma, e_{\mathcal{B}(b)}$$

$$\frac{\eta, \Gamma, e \to \eta', \Gamma', e'}{\eta, \Gamma, \alpha(e) \to \eta', \Gamma', \alpha(e')} \qquad \frac{\alpha \in \Gamma(r)}{\eta, \Gamma, \alpha(r) \to \eta\alpha(r), \Gamma, *} \qquad \frac{\alpha \notin \Gamma(r)}{\eta, \Gamma, \alpha(r) \to \eta, \Gamma, \mathit{fail}_\kappa}$$

$$\frac{\eta, \Gamma, e \to \eta', \Gamma', e' \quad \eta' \models \varphi}{\eta, \Gamma, \varphi[e] \to \eta', \Gamma', \varphi[e']} \qquad \frac{\eta \models \varphi}{\eta, \Gamma, \varphi[v] \to \eta, \Gamma, v} \qquad \frac{\eta, \Gamma, e \to \eta', \Gamma', e' \quad \eta' \not\models_r \varphi}{\eta, \Gamma, \varphi[e] \to \eta, \Gamma, \mathit{fail}_{\varphi(r)}}$$

$$\eta, \Gamma, \texttt{new}\ x : \gamma\ \texttt{in}\ e \to \eta\, \mathit{new}_{\kappa(\gamma)}(r), \Gamma \cup \{\gamma/r\}, e\{r/x\} \qquad \text{if } \kappa(\gamma) \neq \emptyset,\ r \text{ fresh}$$

An access $\alpha(r)$ can be executed if the capabilities associated with $r$ include the action $\alpha$, otherwise it generates a failure. A new resource $r$ is created through the primitive $\texttt{new}\ x : \gamma\ \texttt{in}\ e$, which binds the scope of the *fresh* name $r$ in $e$, and extends the capability environment $\Gamma$. For conditionals, we assume as given a total function $\mathcal{B}$ that evaluates the boolean guards. An expression $\varphi[e]$ can evolve to $\varphi[e']$, provided that the resulting history $\eta'$ satisfies all the relevant instantiations $\varphi(r)$. A failure $\mathit{fail}_{\varphi(r)}$ occurs when, for some resource $r$, $\varphi(r)$ is violated by the extended history $\eta'$. Formally, let $\eta|_\varphi$ be the (longest) subsequence of $\eta$ containing only the accessess $\alpha(r)$ such that $\alpha \in \ell(\varphi)$, and let $R$ be the set of resources mentioned in $\eta|_\varphi$. We say that a history $\eta$ obeys a policy $\varphi$, in symbols $\eta \models \varphi$, when $\eta|_\varphi \in \mathcal{L}(\varphi(r, R))$ for each $r \in R$. Similarly, we write $\eta \not\models_r \varphi$ when $\eta|_\varphi \notin \mathcal{L}(\varphi(r, R))$ for some $r \in R$.

*Example 2.* Let $\eta = \mathit{new}_{\mathsf{File}}(r_0)\, \mathit{open}(r_0)\, \mathit{write}(r_0)\, \mathit{close}(r_0)\, \mathit{open}(r_1)\, \mathit{read}(r_1)\, \mathit{connect}$, and let $\varphi, \varphi'$ be the file usage and Chinese Wall policies of Ex.1. Then, $\eta \models \varphi$, because $\eta|_\varphi = \mathit{open}(r_0)\mathit{write}(r_0)\mathit{close}(r_0)\mathit{open}(r_1)\mathit{read}(r_1)$, $R = \{r_0, r_1\}$ and so $\eta|_\varphi \in \mathcal{L}(\varphi(r_0, R)) \cap \mathcal{L}(\varphi(r_1, R))$. Instead, $\eta \not\models_{r_1} \varphi'$, because $\eta|_{\varphi'} = \mathit{new}_{\mathsf{File}}(r_0)\mathit{read}(r_1)\mathit{connect} \notin \mathcal{L}(\varphi'(r_1, R))$.

## 3 Static semantics

We statically predict the histories generated by programs at run-time through a type and effect system, building upon [1, 18]. The types extend those of the implicitly-typed $\lambda$-calculus, and the effects are *history expressions*, which over-approximate the aspects of the program behaviour that are relevant for resource usage. History expressions include the empty $\varepsilon$, events $\alpha(\rho)$, where $\rho \in \mathsf{Res} \cup \mathsf{Nam} \cup \{?\}$ ($n, n', \ldots \in \mathsf{Nam}$ are *names*, to be instantiated to fresh resources, and ? is a wildcard for all names), resource binding $\nu n.H$, sequencing $H \cdot H'$, non-deterministic choice $H + H'$, policy framing $\varphi[H]$, and recursion $\mu h.H$, where $\mu$ binds the occurrences of $h$ in $H$.

***Histories.*** The intended meaning of a history expression is a set of histories, extended to keep track of the policy framings through the special *framing events* $[_\varphi$ and $]_\varphi$, that stand respectively for opening and closing the scope of the policy $\varphi$. For example, an (extended) history $\alpha[_\varphi \alpha']_\varphi$ represents a computation that (i) generates an access $\alpha$, (ii) enters the scope of a framing $\varphi[\cdots]$, (iii) generates $\alpha'$ within the scope of $\varphi$, and (iv) leaves the scope of $\varphi$. Note that histories with no framing events were enough to give the operational semantics of $\lambda^{[]}$, where the role of framing events is played by framed expressions. Hereafter, a history may end with the truncation marker $! \notin \mathsf{Act}$. The history $\eta!$ represents a prefix of a possibly non-terminating computation that generates the sequence of events $\eta$. We assume that histories are undistinguishable after truncation, i.e. $\eta!$ followed by $\eta'$ equals to $\eta!$. A history $\eta$ is *balanced* when either $\eta$ is empty, or $\eta$ is an access event, or $\eta = !$, or $\eta = [_\varphi \eta' ]_\varphi$ with $\eta'$ balanced, or $\eta = \eta'\eta''$ with both $\eta'$ and $\eta''$ balanced. For example, $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_{\varphi'}]_\varphi$ is balanced, while $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_\varphi$ is not. In what follows, we will only consider *well-formed* histories that are prefixes of some balanced history. Non well-formed histories, like e.g. $]_\varphi \alpha$, are not interesting, because they do not correspond to any $\lambda^{[]}$ computation.

The denotation of $H = (\nu n.\, \alpha(n)) \cdot (\nu n.\, \alpha(n))$ will contain *all* the histories $\alpha(r)\alpha(r')$ for $r \neq r'$. To this purpose we introduce *template histories* $\nabla \boldsymbol{n}.\, \eta$, where $\eta$ may possibly contain events of the form $\alpha(n)$, and $\nabla$ acts as a binder of the names in the finite set $\boldsymbol{n}$. Back to our example, the semantics of $H$ is rendered by $\nabla n, n'.\, \alpha(n)\alpha(n')$. Bound names in template histories are $\alpha$-convertible. We write $\eta$ for $\nabla\emptyset.\, \eta$, and $\nabla \boldsymbol{n}.\, \nabla \boldsymbol{m}.\, \eta$ for $\nabla \boldsymbol{nm}.\, \eta$. A template history $\nabla \boldsymbol{n}.\, \eta$ is balanced when $\eta$ is such. Let $\mathcal{H}, \mathcal{H}'$ range over sets of balanced template histories (BTH for short) and let $N(\eta)$ be the set of names occurring in $\eta$. The set $\varphi[\mathcal{H}]$ denotes $[_\varphi \mathcal{H} ]_\varphi$. Also, we denote with $\mathcal{H}\mathcal{H}'$ the set:

$$\{\, \nabla \boldsymbol{nm}.\, \eta\eta' \mid \nabla \boldsymbol{n}.\, \eta \in \mathcal{H}, \nabla \boldsymbol{m}.\, \eta' \in \mathcal{H}',\ \boldsymbol{n} \cap N(\eta') = \emptyset = \boldsymbol{m} \cap N(\eta) \,\}$$

For example, since $\nabla n.\, \beta(n)$ can be $\alpha$-converted to $\nabla m.\, \beta(m)$, then:

$$(\nabla n.\, \alpha(n))\,(\nabla n.\, \beta(n)) \;=\; (\nabla n.\, \alpha(n))(\nabla m.\, \beta(m)) = \nabla n, m.\, \alpha(n)\beta(m)$$

***Semantics of history expressions.*** The *denotational semantics* $[\![H]\!]_\chi$ of history expressions maps $H$ to a set $\mathcal{H}$ of BTH, in an environment $\chi$ that maps variables $h$ to sets of BTH. We assume that a truncated history always denotes all its truncated prefixes, i.e.,

whenever $\nabla \boldsymbol{n}.\, \eta\eta'! \in \mathcal{H}$, then $\nabla \boldsymbol{n}.\, \eta! \in \mathcal{H}$.

$$[\![\varepsilon]\!]_\chi = \{\varepsilon\} \qquad [\![\alpha(r)]\!]_\chi = \{\alpha(r)\} \qquad [\![\alpha(?)]\!]_\chi = \{\alpha(?)\} \qquad [\![\nu n.\, H]\!]_\chi = \nabla n.\, [\![H]\!]_\chi$$
$$[\![\varphi[H]]\!]_\chi = \varphi[[\![H]\!]_\chi] \qquad [\![H \cdot H']\!]_\chi = [\![H]\!]_\chi\, [\![H']\!]_\chi \qquad [\![H + H']\!]_\chi = [\![H]\!]_\chi \cup [\![H']\!]_\chi$$
$$[\![h]\!]_\chi = \chi(h) \qquad [\![\mu h.H]\!]_\chi = \bigcup_{k>0} f^k(!) \quad \text{where } f(X) = [\![H]\!]_{\chi\{X/h\}}$$

*Example 3.* Let $H_0 = \mu h.\, \alpha \cdot h$, let $H_1 = \mu h.\, h \cdot \alpha$, and let $H_2 = \mu h.\, \nu n.\, \alpha(n) \cdot h$. Then, $[\![H_0]\!]_\emptyset = \alpha^*!$, i.e. $H_0$ generates histories with an arbitrary number of $\alpha$, and never terminates. Instead, $[\![H_1]\!]_\emptyset = \{!\}$, i.e. $H_1$ loops forever, without generating events. The semantics of $H_2$ consists of all the histories $\nabla n_1, \ldots, n_k.\, \alpha(n_1) \cdots \alpha(n_k)!$, for $k > 0$. □

***Equational theory.*** History expressions enjoy some equational properties. Intuitively, the equation $H = H'$ implies that $[\![H]\!]_\chi = [\![H']\!]_\chi$ for all $\chi$. The operation $+$ is associative, commutative and idempotent; $\cdot$ is associative, has the identity $\varepsilon$, and distributes over $+$. The binders $\nu n$ and $\mu h$ can be rearranged, and $\mu h$ can be introduced/eliminated when $h$ does not occur free. The $\nu$ binder can be extruded when it does not bind free names (as usual, $n$ is free in $H$ if it is not in the scope of a $\nu n$, otherwise it is bound). Note that $\nu n$ cannot be always lifted to the top-level: e.g., $\mu h.\nu n.H \neq \nu n.\mu h.H$ in general, because the leftmost history expression represents a loop that creates a new resource at each iteration, while in the rightmost one the new resource is created just before entering the loop. The last two rules allow for introduction/elimination of name binders, and for $\alpha$-conversion. The set $N(H)$ denotes the names in $H$.

$$H + H = H \qquad (H + H') + H'' = H + (H' + H'') \qquad H + H' = H' + H$$
$$(H \cdot H') \cdot H'' = H \cdot (H' \cdot H'') \qquad \varepsilon \cdot H = H = H \cdot \varepsilon$$
$$H \cdot (H' + H'') = H \cdot H' + H \cdot H'' \qquad (H + H') \cdot H'' = H \cdot H'' + H' \cdot H''$$
$$\nu n.\, \nu n'.\, H = \nu n'.\nu n.H \qquad \mu h.\, \mu h'.\, H = \mu h'.\, \mu h.\, H \qquad \varphi[\nu n.\, H] = \nu n.\, \varphi[H]$$
$$\nu n.(H \cdot H') = (\nu n.H) \cdot H' \;\text{ if } n \notin fn(H') \qquad \nu n.(H \cdot H') = H \cdot (\nu n.H') \;\text{ if } n \notin fn(H)$$
$$\nu n.(H + H') = (\nu n.H) + H' \;\text{ if } n \notin fn(H') \qquad \mu h.H = H\{\mu h.H/h\}$$
$$\nu n.H = H \;\text{ if } n \notin fn(H) \qquad \nu n.H = \nu m.H\{m/n\} \;\text{ (capture-avoiding)}$$

Note that we could replace the two constructs $\nu n.H$ and $\mu h.H$ with a single construct $\mu h.\nu n.H$, so defining a *standard form* for history expressions. For example, $\mu h.\, (\varepsilon + \nu n.\, \nu n'.\, \alpha(n) \cdot h \cdot \alpha(n'))$ can be rewritten as $\mu h.\nu n.\, \mu h'.\nu n'.\, \varepsilon + \alpha(n) \cdot h \cdot \alpha(n')$.

***Unbound history expressions.*** Unbound history expressions are history expressions without $\nu$-binders. Binding names in unbound history expressions is driven by the events *new*. For instance, in the unbound $H = new(n) \cdot \alpha(n) + new(m) \cdot \beta(m)$ the event $new(n)$ binds the name $n$, while $new(m)$ binds $m$, i.e. $H$ is "bindified" to the history expression $(\nu n.new(n) \cdot \alpha(n)) + (\nu m.new(m) \cdot \beta(m))$.

Unbound history expressions have an equational theory $\approx$, which is a subtheory of the relation $=$ on history expressions. In particular, the last three equations (folding/unfolding, introduction/elimination of $\nu$ and $\alpha$-conversion) are not permitted on unbound history expressions. Also, the right-distributivity of $\cdot$ over $+$ has the side condition $fn((H + H') \cdot H'') = fn(H \cdot H'' + H' \cdot H'')$. The definition of bound and free names in

unbound history expressions slightly differs from the standard one: $bn(new(n)) = \{n\}$, $fn(new(n)) = \emptyset$, $fn(H \cdot H') = fn(H) \cup (fn(H') \setminus bn(H))$. We also define the set $rn(H) = \{\, n \mid H \approx C(\mu h.\, \nu n.\, H'), h \in fv(H') \,\}$ of *recursive names* in $H$, where $C(\bullet)$ is a context.

To obtain a history expression from an unbound one, we will now introduce the *bindify* transformation $\omega$. This transformation will insert the $\nu$ binders at the right points, provided that the introduced scopes of names do not interfere dangerously. For instance, $\omega$ is undefined on the unbound history expression $H = new(n) \cdot new(n) \cdot \alpha(n)$, because it is unclear whether the action $\alpha$ is performed on the resource $n$ created first or the second *new*). It is then not sound choosing the above $H$ to approximate the histories of e.g. $e = \texttt{new}\ x\ \texttt{in}\ \texttt{new}\ y\ \texttt{in}\ \alpha(y)$, because $new(r)new(r')\alpha(r')$ is not represented by $H$.

$$\omega(\alpha(\rho)) = \alpha(\rho) \quad \text{if } \alpha \neq new \qquad \omega(new(n)) = \nu n.\, new(n) \qquad \omega(h) = h$$

$$\omega(H \cdot H') = \omega(H) \odot \omega(H') \quad \text{if } bn(H) \cap bn(H') = \emptyset$$

$$\omega(H + H') = \omega(H) \oplus \omega(H') \qquad \omega(\mu h.\, H) = \mu h.\, \omega(H) \qquad \omega(\varphi[H]) = \varphi[\omega(H)]$$

$$H \odot H' = \begin{cases} \nu n.\, (\bar{H} \odot H') & \text{if } H \approx \nu n.\, \bar{H}, n \notin rn(H) \\ H \cdot H' & \text{if } rn(H) \cap fn(H') = \emptyset \end{cases}$$

$$H \oplus H' = \begin{cases} \nu n.\, (\bar{H} \oplus \bar{H}') & \text{if } H \approx \nu n.\, \bar{H}, H' \approx \nu n.\, \bar{H}', n \notin rn(H + H') \\ H + H' & \text{otherwise} \end{cases}$$

The event $new(n)$ drives the introduction of the actual binder $\nu n$ in history expressions: the scope of $n$ in $H$ is entered just before the $new(n)$, and it is left as soon as needed no longer, e.g. $new(n) \cdot (\mu h.\, \varepsilon + new(n') \cdot \alpha(n) \cdot \alpha(n') \cdot h) \cdot \alpha(n)$ is bindified into $(\nu n.\, new(n) \cdot (\mu h.\, \varepsilon + (\nu n'.\, new(n') \cdot \alpha(n) \cdot \alpha(n') \cdot h)) \cdot \alpha(n))$. Instead, $\omega$ is not defined on $(\mu h.\, \varepsilon + new(n) \cdot h) \cdot \alpha(n)$, because the name $n$ accessed through $\alpha$ could be any name generated by the *new* inside the loop.

***Type & Effect system.*** We define below a type and effect system for $\lambda^{[\,]}$. Effects $H$ are unbound history expressions. Types $\tau$ comprise the unit $\mathbf{1}$, sets $R \subseteq (\mathsf{Res} \cup \mathsf{Nam}) \times 2^{\mathsf{Act}}$, and arrows $\tau \xrightarrow{H} \tau$. For instance, a resource $r$ with capabilities $\gamma$ has the singleton type $\{(r, \gamma)\}$ (we omit the capabilities when irrelevant). Type environments have the form $\Delta;\, \xi : \tau$ where $\xi \in \mathsf{Var} \cup \mathsf{Res}$ is not already in $dom(\Delta)$. A typing judgment $\Delta \vdash e : \tau \triangleright H$ means that, in a type environment $\Delta$, the expression $e$ evaluates to a value of type $\tau$, and produces a history belonging to the effect $H$. In the functional type $\tau \xrightarrow{H} \tau'$, $H$ describes the *latent* effect associated with an abstraction, i.e. one of the histories represented by $H$ will be generated when such an abstraction is applied to a value.

To keep our type system as simple as possible, and still allowing to deal with the *escape* of freshly created resources, we avoid to explicitly introduce binders on types. Instead, we have raised the action *new* to the key role of an implicit binder. E.g., the type $\mathbf{1} \xrightarrow{new(n) \cdot \alpha(n)} \{n\}$ is for a function that generates a fresh resource upon each invocation, accesses it through the action $\alpha$, and then returns it. The bindify transformation, together with some side-conditions on the typing rules, ensure that the typing derivation do not exploit the absence of explicit binders to identify names that should be kept distinct. As a global invariant on typing derivations, we require that in a type & effect $\tau \triangleright H$, the bound names of $H$ are disjoint from those of $\tau$ (i.e. the bound names in latent effects).

The relation $\sqsubseteq$ is used to define subtypes and subeffects. Roughly, $H \sqsubseteq H'$ means that $[\![\omega(H)]\!] \subseteq [\![\omega(H')]\!]$ — when both $\omega(H)$ and $\omega(H')$ are defined and closed. For instance, $C(H) \sqsubseteq C(H + H')$, in any context $C(H)$. The relation $\sqsubseteq$ comprises a version of folding/unfolding that creates *fresh* names upon unfolding (so not to prevent from bindification). For example, if $H = \mu h.\, new(n) \cdot \alpha(n) \cdot h$, then $new(n') \cdot \alpha(n') \cdot H \sqsubseteq H$. Subtypes are defined as usual, contravariant in the argument type and covariant in the return type (the latent effect is invariant).

$$\mathbf{1} \sqsubseteq \mathbf{1} \quad R \sqsubseteq R' \text{ if } R \subseteq R' \quad \{(n,\gamma)\} \cup R \sqsubseteq \{(?,\gamma)\} \cup R \quad \tau_0 \xrightarrow{H} \tau_0' \sqsubseteq \tau_1 \xrightarrow{H} \tau_1' \text{ if } \begin{array}{l} \tau_0 \sqsupseteq \tau_1 \\ \tau_0' \sqsubseteq \tau_1' \end{array}$$

$$H \sqsubseteq H' \text{ if } H \approx H' \quad H \sqsubseteq H + H' \quad H\theta\{\mu h.\, H/h\} \sqsubseteq \mu h.\, H \quad (\theta \text{ maps } bn(H) \text{ into fresh names})$$

$$C(H) \sqsubseteq C(H') \quad \text{if } H \sqsubseteq H',\ (bn(H) \setminus bn(H')) \cap N(C) = \emptyset,\ fn(C(H)) \subseteq fn(C(H'))$$

We now introduce the type and effect system for $\lambda^{[]}$. An access $\alpha(e)$ has type $\mathbf{1}$, provided that the type of $e$ is a set of resources $R$, and each resource in $R$ has the capability $\alpha$. The effect of $\alpha(e)$ can be any of the accesses $\alpha(\rho)$ for $(\rho,\gamma) \in R$. 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 side condition ensures that the free names in the effect of the argument are not captured by the effect of the function. The actual effect of an abstraction is the empty history expression, while its latent effect is equal to the actual effect of the function body. Note that the rule for abstraction constrains the premise to equate the actual and latent effects. A resource creation generates a fresh name, and binds it in the effect through the event *new*. The last two rules allow for *weakening* of types/effects and $\alpha$-conversion, respectively. The side condition † on weakening requires that names created through subeffecting are disjoint from the names in the type (e.g. the weakening $1 \xrightarrow{\alpha(n)} 1 \rhd \varepsilon \sqsubseteq 1 \xrightarrow{\alpha(n)} 1 \rhd \varepsilon + new(n)$ is not permitted, because the *new* event would capture the free $n$ in the type). Although $\alpha$-conversion is not permitted on unbound history expressions, we allow it on types, e.g. $\mathbf{1} \xrightarrow{new(n)\cdot\alpha(n)} \{n\}$ can be $\alpha$-converted to $\mathbf{1} \xrightarrow{new(m)\cdot\alpha(m)} \{m\}$.

$$\frac{\Delta \vdash e : R \rhd H \quad \forall(\rho,\gamma) \in R.\ \alpha \in \gamma}{\Delta \vdash \alpha(e) : \mathbf{1} \rhd H \cdot \Sigma_{(\rho,\gamma)\in R}\ \alpha(\rho)} \qquad \frac{\xi : \tau \in \Delta}{\Delta \vdash \xi : \tau \rhd \varepsilon} \qquad \frac{\Delta; x : \tau; z : \tau \xrightarrow{H} \tau' \vdash e : \tau' \rhd H}{\Delta \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau' \rhd \varepsilon}$$

$$\frac{\Delta \vdash e : \tau \xrightarrow{H''} \tau' \rhd H \quad \Delta \vdash e' : \tau \rhd H'}{\Delta \vdash e\, e' : \tau' \rhd H \cdot H' \cdot H''}\ bn(H) \cap fn(H') = \emptyset \qquad \frac{\Delta \vdash e : \tau \rhd H}{\Delta \vdash \varphi[e] : \tau \rhd \varphi[H]}$$

$$\frac{\Delta; x : \{(n,\gamma)\} \vdash e : \tau \rhd H \quad \kappa(\gamma) \neq \emptyset}{\Delta, \vdash \mathtt{new}\ x : \gamma\ \mathtt{in}\ e : \tau \rhd new_{\kappa(\gamma)}(n) \cdot H}\ \begin{array}{l} n \notin \Delta \\ n \notin bn(\tau) \\ n \notin bn(H) \end{array} \qquad \frac{\Delta \vdash e : \tau \rhd H \quad \Delta \vdash e' : \tau \rhd H}{\Delta \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau \rhd H}$$

$$\frac{\Delta \vdash e : \tau \rhd H}{\Delta \vdash e : \tau' \rhd H'}\ \begin{array}{l} \tau \sqsubseteq \tau' \\ H \sqsubseteq H' \end{array}{}^{\dagger} \qquad \frac{\Delta \vdash e : \tau \xrightarrow{H} \tau' \rhd H'}{\Delta \vdash e : \tau \xrightarrow{H\theta} \tau'\theta \rhd H'}\ \begin{array}{l} dom(\theta) \cap bn(H') = \emptyset \\ \theta \text{ capture-avoiding} \end{array}$$

$$\dagger\ \ (bn(H) \setminus bn(H')) \cap N(\tau) = \emptyset = (bn(H') \setminus bn(H)) \cap N(\tau')$$

*Example 4.* We have the following typing judgements (see App. **??** for details):

$$\begin{array}{l}
\emptyset \vdash \alpha(\texttt{new}\ x : \gamma\ \texttt{in}\ \texttt{new}\ y : \gamma'\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y) : \mathbf{1} \\
\quad \triangleright H_1 = new_{\kappa(\gamma)}(n) \cdot new_{\kappa(\gamma')}(n') \cdot (\alpha(n) + \alpha(n')) \quad \text{if } \alpha \in \gamma \cap \gamma' \\
\quad \omega(H_1) = \nu n.\ \nu n'.\ new_{\kappa(\gamma)}(n) \cdot new_{\kappa(\gamma')}(n') \cdot (\alpha(n) + \alpha(n')) \\
\emptyset \vdash \texttt{let}\ f = (\lambda x.\texttt{new}\ n : \gamma\ \texttt{in}\ \alpha(n); n)\ \texttt{in}\ \alpha'(f*; f*) : \mathbf{1} \\
\quad \triangleright H_2 = new_{\kappa(\gamma)}(n) \cdot \alpha(n) \cdot new_{\kappa(\gamma)}(n') \cdot \alpha(n') \cdot \alpha'(n') \quad \text{if } \alpha, \alpha' \in \gamma \\
\quad \omega(H_2) = (\nu n.\ new_{\kappa(\gamma)}(n) \cdot \alpha(n)) \cdot (\nu n'.\ new_{\kappa(\gamma)}(n') \cdot \alpha(n') \cdot \alpha'(n')) \\
\emptyset \vdash \texttt{let}\ g = (\texttt{new}\ n : \gamma\ \texttt{in}\ \lambda x.\ \alpha(n); n)\ \texttt{in}\ \alpha'(g*; g*) : \mathbf{1} \\
\quad \triangleright H_3 = new_{\kappa(\gamma)}(n) \cdot \alpha(n) \cdot \alpha(n) \cdot \alpha'(n) \quad \text{if } \alpha, \alpha' \in \gamma \\
\quad \omega(H_3) = \nu n.\ new_{\kappa(\gamma)}(n) \cdot \alpha(n) \cdot \alpha(n) \cdot \alpha'(n) \\
\emptyset \vdash (\lambda_z x.\texttt{new}\ n : \gamma\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ \alpha(n)\ \texttt{else}\ \alpha'(n); zx) * : \mathbf{1} \\
\quad \triangleright H_4 = \mu h.\ new_{\kappa(\gamma)}(n) \cdot (\alpha(n) + \alpha'(n) \cdot h) \quad \text{if } \alpha, \alpha' \in \gamma \\
\quad \omega(H_4) = \mu h.\ \nu n.\ new_{\kappa(\gamma)}(n) \cdot (\alpha(n) + \alpha'(n) \cdot h) \\
\emptyset \vdash \alpha((\lambda_z x.\texttt{new}\ n : \gamma\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ n\ \texttt{else}\ \alpha'(n); zx) *) : \mathbf{1} \\
\quad \triangleright H_5 = (\mu h.\ new_{\kappa(\gamma)}(n) \cdot (\varepsilon + \alpha'(n) \cdot h)) \cdot \alpha(?) \quad \text{if } \alpha, \alpha' \in \gamma \\
\quad \omega(H_5) = (\mu h.\ \nu n.\ new_{\kappa(\gamma)}(n) \cdot (\varepsilon + \alpha'(n) \cdot h)) \cdot \alpha(?)
\end{array}$$

***Type safety.*** Let $\eta = \beta_1\beta_2\cdots$ be a history. We define $\eta^\flat$ as the history obtained from $\eta$ by erasing all the framing events, and $\eta^\partial$ as the set of all the prefixes of $\eta$, without !. E.g., $(\alpha\alpha'[_\varphi\alpha'']^\flat_\varphi)^\partial = (\alpha\alpha'\alpha'')^\partial = \{\varepsilon, \alpha, \alpha\alpha', \alpha\alpha'\alpha''\}$. Let $\Delta_0$ comprise $r : \{(r,\gamma)\}$ whenever $(r,\gamma) \in \Gamma_0$. Our type and effect system correctly approximates the actual run-time histories; as usual, precision is lost with conditionals and with recursive functions. Also, you may lose the identity of names exported by recursive functions (see $H_5$ above).

**Theorem 1.** *Let $\Delta_0 \vdash e : \tau \triangleright H$, $\omega(H)$ closed, and $\varepsilon, \Gamma_0, e \to^* \eta, \Gamma, e'$. Then, $\eta \in [\![\omega(H)]\!]^{\flat\partial}$.*

A *valid* history does not violate any resource usage constraint. Consider the security policy $\varphi'$ of Ex.1: the history $\eta = open(r)read(r)\varphi'[connect]$ is *not* valid, because the *connect* occurs within a framing enforcing $\varphi'$, and $open(r)read(r)connect$ does not obey $\varphi'$. To formally define validity, we introduce the notion of safe-sets. The history $\eta$ above has one safe-set: $\varphi'[\{open(r)read(r), open(r)read(r)connect\}]$, meaning that the scope of the framing $\varphi'[\cdots]$ encloses the two histories within the curly brackets. To have a short, inductive definition of the safe-sets $S(\eta)$ we first balance all the framings of $\eta$, e.g. $[_\varphi\alpha$ becomes $[_\varphi\alpha]_\varphi = \varphi[\alpha]$. Then, we define:

$$S(\varepsilon) = \emptyset \qquad S(\eta\,\alpha(r)) = S(\eta) \qquad S(\eta_0\,\varphi[\eta_1]) = S(\eta_0\,\eta_1) \cup \varphi[\eta_0^\flat\,(\eta_1^\flat)^\partial]$$

A history $\eta$ is *valid* when $\varphi[\mathcal{H}] \in S(\eta)$ implies $\eta' \models \varphi$ for all $\eta' \in \mathcal{H}$. Note that past events cannot be hidden, because policy framings can always inspect the whole past history. For example, a history $\alpha_1\varphi[\alpha_2]\alpha_3$ is valid when $\alpha_1 \models \varphi$ and $\alpha_1\alpha_2 \models \varphi$ (even if $\alpha_1$ is outside of the safety framing), while $\alpha_1\alpha_2\alpha_3$ is not required to satisfy $\varphi$ any longer. A history expression $H$ is *valid* when all the histories in $[\![H]\!]$ are such. Our type and effect system guarantees the following type safety property.

**Theorem 2 (Type Safety).** *Let $\Delta_0 \vdash e : \tau \triangleright H$. Then:*

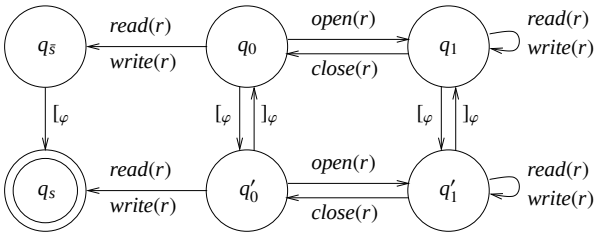
*(i)* $\varepsilon, \Gamma_0, e \twoheadrightarrow^* \eta, \Gamma, fail_\kappa$ *(e respects capabilities)*
*(ii) if $\omega(H)$ is valid, then* $\varepsilon, \Gamma_0, e \twoheadrightarrow^* \eta, \Gamma, fail_{\varphi(r)}$ *(e respects the prescribed usages).*

## 4 Static verification

We verify the validity of history expressions by model-checking Basic Process Algebras (BPAs) with finite state automata. The standard decision procedure for verifying that a BPA process $p$ satisfies a regular property $\varphi$ amounts to constructing the pushdown automaton for $p$ and the automaton for $\neg\varphi$. Then, the property holds if the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton, is empty. This problem is known to be decidable, and several algorithms and tools show this approach feasible [9].

A first problem, solved in [1], is that the arbitrary nesting of framings makes validity of histories a non-regular, property. For example, $[\![\mu h.\, \alpha + h \cdot h + \varphi[h]]\!]$ denotes histories with unbounded pairs of balanced $[_\varphi$ and $]_\varphi$, so it is a context-free, non-regular language. In [1] we defined a *regularization* $\downarrow$ of history expressions such that $H$ is valid if and only if each $\eta \in [\![H\downarrow]\!]$ satisfies $\varphi_{[\,]}$. The formula $\varphi_{[\,]}$ is defined through the automaton $A_{\varphi_{[\,]}(x)}$, a smooth transformation of $A_{\varphi(x)}$ taking into account entering/leaving the frame $\varphi[\cdots]$. Hereafter, we assume that history expressions have been regularized (a simple extension of [1] suffices).

*Example 5.* The framed version of the file usage policy $\varphi(r)$ of Ex.1 is described by the automaton $A_{\varphi_{[\,]}(r)}$ below. The top (resp. bottom) layer models being outside (resp. inside) the scope of $\varphi$. All states have self-loops (not displayed in the figure) for the irrelevant events. For instance, the history $[_\varphi open(r)\, close(r)\, read(r)$ is not accepted.




A second problem, solved here, is that now history expressions may create new names, while BPAs cannot handle fresh names. Verifying validity would thus need to check an unbounded set of policies $\varphi(r)$, e.g. the histories denoted by $H = \varphi[\mu h.\, \varepsilon + \nu n.\alpha(n) \cdot h]$ must satisfy all the policies $\varphi(r_0), \varphi(r_1), \ldots$ for each fresh resource created within the loop. Thus, we would have to intersect an infinite number of finite state automata to verify $H$ valid, which is unfeasible. As a first contribution, we extract from a history expression $H$ a BPA and a *finite* set of usage policies, that suffice for verifying $H$ valid. The intuition is that a new resource $r$ created under a $\mu h$ lives for a *single* iteration of the loop, and in the other iterations we do not care of the actual resources generated

(therefore we denote with _ these "dummy" resources). Formally, the function $\mathcal{M}(H)_\Theta$ takes as input a history expression $H$ and a function $\Theta$ from history variables $h$ to BPA variables $X$. Its output is a guarded BPA process $p$, a finite set of definitions $\Delta$ for BPA variables, and a finite set of usage policies $\Pi$. Without loss of generality, we assume that $H$ is regularized, and that its variables are all distinct.

$$\mathcal{M}(\varepsilon)_\Theta = \langle \varepsilon, \emptyset, \emptyset \rangle \qquad \mathcal{M}(h)_\Theta = \langle \Theta(h), \emptyset, \emptyset \rangle$$

$$\mathcal{M}(\alpha(\zeta))_\Theta = \langle \alpha(\zeta), \emptyset, \{\, \varphi(\zeta) \mid \alpha \in \ell(\varphi) \wedge \kappa(\{\alpha\}) = \kappa(\varphi) \,\} \rangle \qquad \zeta \in \mathsf{Res} \cup \{?\}$$

$$\mathcal{M}(H_0 \cdot H_1)_\Theta = \langle p_0 \cdot p_1, \Delta_0 \cup \Delta_1, \Pi_0 \cup \Pi_1 \rangle, \text{ where } \mathcal{M}(H_i)_\Theta = \langle p_i, \Delta_i, \Pi_i \rangle$$

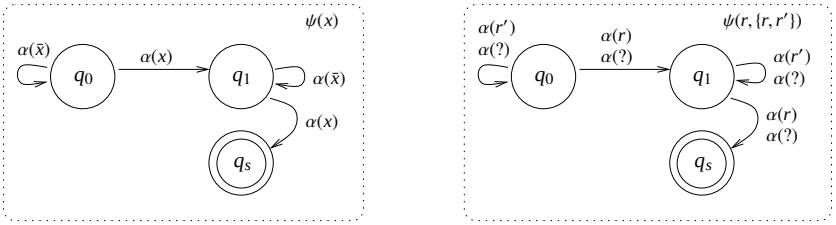
$$\mathcal{M}(H_0 + H_1)_\Theta = \langle p_0 + p_1, \Delta_0 \cup \Delta_1, \Pi_0 \cup \Pi_1 \rangle, \text{ where } \mathcal{M}(H_i)_\Theta = \langle p_i, \Delta_i, \Pi_i \rangle$$

$$\mathcal{M}(\varphi[H])_\Theta = \langle [_\varphi \cdot p \cdot ]_\varphi, \Delta, \Pi \rangle, \text{ where } \mathcal{M}(H)_\Theta = \langle p, \Delta, \Pi \rangle$$

Access events, variables, concatenation and choice are mapped into the corresponding BPA counterparts. An expression $\varphi[H]$ is mapped to the BPA for $H$, surrounded by the opening and closing events of the $\varphi$-framing. The tricky case is that of recursion and new name generation, not shown above (the items $\Delta$ and $\Theta$ will indeed be populated and exploited in the recursive case). We shall outline, with the help of the following examples, the stages that lead to the correct definition of $\mathcal{M}(H)$ in such cases.

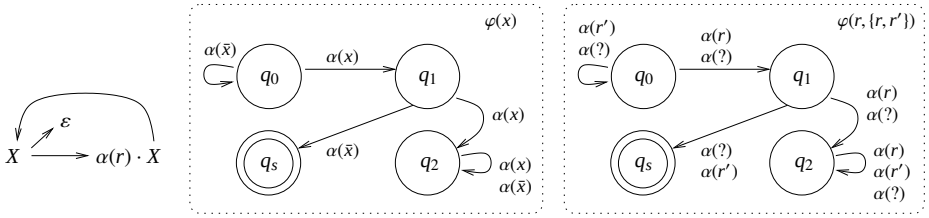
The component $\Pi$ in $\mathcal{M}(H)$ contains the set of all the usage policies that are needed in verifying the validity of $H$. Let $R$ be the (finite) set of resources in the BPA of $\mathcal{M}(H)$. For each event $\alpha(r)$ contained in $\mathcal{M}(H)$ (for $r \neq \_$), the set $\Pi$ comprises all the policies $\varphi(r, R)$ such that the kind of $\varphi$ is consistent with that of $\alpha$ (i.e. $\kappa(\varphi) = \kappa(\{\alpha\})$, and $A_{\varphi(x)}$ has some edge labelled with $\alpha$ (i.e. $\alpha \in \ell(\varphi)$).

*Example 6.* Consider the history expression $H = \nu n.\, \nu n'.\, \alpha(n) \cdot \alpha(n') \cdot \alpha(?)$. Then, a sound BPA for $\mathcal{M}(H)$ is $\alpha(r) \cdot \alpha(r') \cdot \alpha(?)$ where $r$ and $r'$ are two distinct resources. For instance, consider a policy $\psi(x)$ saying that the action $\alpha$ cannot be performed twice on the same resource (left-hand side of the figure below).



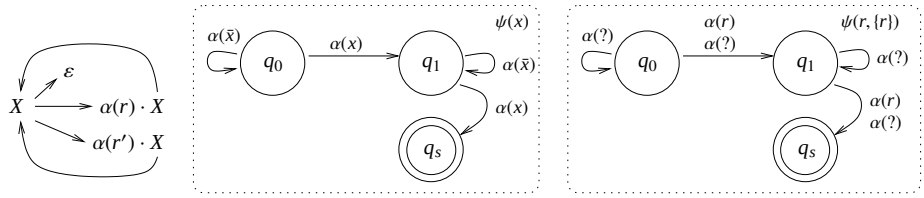

Clearly, the above-mentioned BPA violates $\psi(r)$ (right-hand side of the figure above), consistently with the fact that the wildcard ? represents any resource, including $r$ (e.g. $\alpha(r)\alpha(r')\alpha(r) \in [\![H]\!]$ violates $\psi$). Instead, the BPA above correctly respects a policy $\psi'$ requiring that $\alpha$ is not executed *three* times on the same resource. So, $\mathcal{M}(H)$ correctly reflects violations and obeyance to the relevant policies. In this sense we can say that $\mathcal{M}(H)$ is sound and complete (see Theorem 4 below).

*Example 7.* Let $H = \mu h.\,(\varepsilon + \nu n.\,\alpha(n) \cdot h)$. A first, naïve solution to obtain $\mathcal{M}(H)$ would be that of picking out a resource $r$, and then modelling the BPA as a recursive process, where at each step the event $\alpha(r)$ is executed (left-hand side of the picture below).



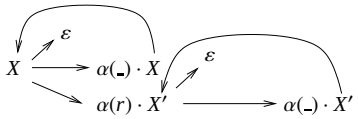

However, this solution is *not* sound. To see why, consider a policy $\varphi(x)$ (modelled by the template usage automaton $A_{\varphi(x)}$ central in the picture above), saying that, for each resource $x$, the first event $\alpha(x)$ is necessarily followed by another $\alpha(x)$. Clearly, $H$ violates the policy (e.g. $\eta = \alpha(r)\alpha(r') \in [\![H]\!]$, and $\eta \not\models \varphi(r, \{r, r'\})$, see the instantiated automaton on the right-hand side of the picture above). Instead, the BPA does *not* violate the policy, and so it is unsound.

As a second try, consider a slight variation of the BPA above (left-hand side of the picture below), where, at each step, one among the events $\alpha(r)$ or $\alpha(r')$ can be executed. This BPA correctly violates $\varphi(r, \{r, r'\})$ above.




Although this solution is sound, it is not complete. Consider for instance the policy $\psi(x)$ saying that the action $\alpha$ cannot be performed twice on the same resource $x$ (see the template automaton in the center of the figure above). Although $H$ obeys $\psi$, the BPA does not: indeed, the BPA trace $\alpha(r)\alpha(r)$ violates $\psi(r, \{r\})$ (the instantiated automaton $A_{\psi(r,\{r\})}$ is depicted in the right-hand side of the picture above).

To recover completeness, we must ensure that the BPA does not execute the same event $\alpha(r)$ twice. To do that, the BPA is composed of two loops: the first loop executes $\alpha(\_)$ on a dummy resouce $\_$, then, the BPA executes $\alpha(r)$ once, and finally the second loop executes $\alpha(\_)$ (see the left part in the figure below). Template automata are only instantiated with the resource $r$ (and not with $\_$).




This solution is both sound and complete. For soundness, the BPA correctly violates $\varphi(r, \{r, \_\})$, e.g. with the trace $\alpha(r)\alpha(\_)$. For completeness, the BPA respects $\psi(r, \{r, \_\})$ (note that here it is important that $\psi(\_)$ is not considered).

More generally, when $H$ is a loop, then $\mathcal{M}(H)$ is a BPA which (i) runs the loop an arbitrary number of times, substituting _ for the actual freshly generated resources, (ii) runs a *single* iteration of the loop, using a unique instantiation of names, and (iii), runs the same loop as (i). Special care is needed to avoid replication of the same names, e.g. in case of nested recursion.

*Example 8.* Let $\varphi_k$ require that no more than $k$ files can be created, i.e. $A_{\varphi_k(x)}$ has states $q_0, \ldots, q_k, q_s$ and edges $q_i \xrightarrow{new_{\mathsf{File}}(\zeta)} q_{i+1}$ for $i \in 0..k-1$ and $q_k \xrightarrow{new_{\mathsf{File}}(\zeta)} q_s$, for $\zeta \in \{x, \bar{x}\}$. Let $H = \varphi[\varphi_k[\mu h.\, \varepsilon + \nu n.\, new_{\mathsf{File}}(n) \cdot open(n) \cdot read(n) \cdot close(n) \cdot h]]$, where $\varphi$ is the file usage policy of Ex.1. Then, $\mathcal{M}(H)_\emptyset = \langle [_\varphi \cdot [_{\varphi_k} \cdot X \cdot ]_{\varphi_k} \cdot ]_\varphi, \Delta, \{\varphi(r), \varphi_k(r)\}\rangle$, where $\Delta$ comprises the following definitions (we abbreviate $new_{\mathsf{File}}$ with $\alpha_n$, *read* with $\alpha_r$, etc.):

$$X \triangleq \varepsilon + \Big(\alpha_n(\_) \cdot \alpha_o(\_) \cdot \alpha_r(\_) \cdot \alpha_c(\_) \cdot X\Big) + \Big(\alpha_n(r) \cdot \alpha_o(r) \cdot \alpha_r(r) \cdot \alpha_c(r) \cdot X'\Big)$$
$$X' \triangleq \varepsilon + \alpha_n(\_) \cdot \alpha_o(\_) \cdot \alpha_r(\_) \cdot \alpha_c(\_) \cdot X'$$

Note that each computation of the BPA $\langle p, \Delta\rangle$ obeys the file usage policy $\varphi(r, \{r, \_\})$, while there exist computations that violate $\varphi_k(r, \{r, \_\})$.

We now state the correspondence between history expressions and BPAs. The prefixes of the histories generated by a history expression $H$ (i.e. $[\![H]\!]^\partial$) are all and only the strings that label the computations of the extracted BPA, after a renaming of resources. A special case is that of ?, which may stand for any resource. To deal with it, we define the "up-to-?" relation $=_?$ between histories: $=_?$ is the least reflexive relation such that, for any $r$, $\eta_0\alpha(r)\eta_1 =_? \eta_0'\alpha(?)\eta_1'$ whenever $\eta_0 =_? \eta_0'$ and $\eta_1 =_? \eta_1'$.

**Theorem 3.** *Let $\mathcal{M}(H)_\emptyset = \langle p, \Delta, \Pi\rangle$. For each $\eta \in [\![H]\!]^\partial$, there exist $\eta' \in [\![p, \Delta]\!]$ and a substitution $\xi$ from $\mathsf{Res}$ to $\mathsf{Res} \cup \{\_\}$ such that $\xi(\eta) =_? \eta'$ ($[\![p, \Delta]\!]$ is the trace semantics). Conversely, for each $\eta \in [\![p, \Delta]\!]$, there exists some $\xi$ and $\eta' \in [\![H]\!]^\partial$ such that $\xi(\eta') =_? \eta$.*

*Example 9.* Let $H = \mu h.\, \nu n.\, \varepsilon + \alpha(n) \cdot h$. Then, the BPA extracted from $H$ is $\langle X, \Delta\rangle$, where $\Delta = \{X \triangleq \varepsilon + \alpha(\_) \cdot X + \alpha(r) \cdot X', X' \triangleq \varepsilon + \alpha(\_) \cdot X'\}$. Let $\eta = \alpha(r_0)\alpha(r_1)\alpha(r_2) \in [\![H]\!]^\partial$, and let $\eta' = \alpha(\_)\alpha(r)\alpha(\_)$ be a string in $[\![X, \Delta]\!]$. If $\xi = \{\_/r_0, r/r_1, \_/r_2\}$, then $\xi(\eta) = \eta'$.

The theorem above enables us to verify the validity of a (regularized) history expression $H$ by extracting $\mathcal{M}(H)_\emptyset = \langle p, \Delta, \Pi\rangle$ and then model-checking the BPA $\langle p, \Delta\rangle$ against the finite set of policies $\Pi$. Indeed, a valid computation of the BPA is recognized by the intersection of the finite state automata $A_{\varphi_{[\,]}(r)}$, for all $\varphi(r)$ in $\Pi$. Together with Theorem 2, a $\lambda^{[\,]}$ expression *never goes wrong* if its effect is checked valid.

**Theorem 4.** *Let $\mathcal{M}(H)_\emptyset = \langle p, \Delta, \Pi\rangle$. Then, $H$ is valid iff $[\![p, \Delta]\!] \models \bigwedge\{\, \varphi_{[\,]}(r) \mid \varphi(r) \in \Pi \,\}$.*

## 5 Related Work and Conclusions

We proposed a novel approach to the resource usage problem, within an extension of the $\lambda$-calculus that features creation/access to resources, and regular usage policies with a local scope. To efficiently enforce policies, we have exploited a two-step static analysis.

We defined a type and effect system that over-approximates the run-time behaviour of a program as a history expression. In spite of the augmented flexibility given by the nesting of policy scopes and by resource creation, we transformed history expressions so that model checking their validity is decidable. Our technique manages to represent the generation of an unbounded number of resources in a finitary manner. Yet, we do not lose the possibility of verifying interesting properties of programs (see Ex. 9). When a history expression is valid, we can safely dispose the execution monitor. Otherwise, the soft-typing approach in [2] allows for substituting local checks for local policies, thus making the dynamic control of accesses efficient. Although our policies can always inspect the whole past history, one can easily limit the scope from the side of the past: it suffices to mark in the history the point in time $\beta_\varphi$ from which checking a policy $\varphi$ has to start; the corresponding automaton discards then all the events before $\beta_\varphi$. Type inference has not been considered here, but we do not see major obstacles in extending [18] to our case. Another research direction consists in extending $\lambda^{[]}$ in a distributed setting, to study secure discovery and composition of services [3].

Many authors [7, 8, 14, 21] mixed static and dynamic techniques to transform programs and make them obey a given global policy. Colcombet and Fradet [7] abstracted a program into an instrumented control flow graph, then minimized and converted back to a program that is guaranteed to abort just before violating the property. Marriot, Stuckey and Sulzmann [14] over-approximated the run-time behaviour of a program through a context-free grammar. A finite-state automaton models the permitted resource usages. If the language generated by the grammar is not included in the language accepted by automaton, the program is instrumented with the local checks and the tracking operations needed to make it obey the policy, similarly to [2]. Our programming model allows for local policies and access events parametrized over dynamically created resources, while [7, 8, 14, 21] only consider global policies and no parametrized events.

Igarashi and Kobayashi [12] extended the $\lambda$-calculus with primitives for creating and accessing resources, and for defining their permitted usage patterns. An execution is resource-safe when the possible patterns are within the permitted ones. A type system guarantees well-typed expressions to be resource-safe. However, they do not present any algorithm to effectively check whether the inferred usages conform to the permitted ones. Instead, here we provided $\lambda^{[]}$ with a static verification technique; clearly, also [12] might be amenable to static verification, if one restricts the language of permitted usages to a decidable subset. Furthermore, the policies of [12] can only speak about the usage of *single* resources, while ours can span over many resources, of different kinds, e.g. the Chinese Wall of Ex. 1.

Skalka and Smith [18] proposed a $\lambda$-calculus with local checks that enforce linear $\mu$-calculus properties [6, 13] on the past history. A type and effect system approximates the possible run-time histories, whose validity can be statically verified by model checking $\mu$-calculus formulae over Basic Process Algebras [5, 9]. Compared to [18], we feature dynamic resource creation, and local policies instead of local checks. On a more concrete level, the same ideas are applied in [19] to define a type and effect system for an extension of Featherweight Java, featuring histories and security checks.

Walker [22] mixed static and dynamic techniques with proof-carrying code [15]. Properties are specified by *security automata* [4, 17]. When a security-unaware program

is compiled, a centralized policy tells where to insert local checks, in order to obtain provably-secure compiled code. An optimization phase follows: whenever a check is removed, it is replaced by a proof that the optimized code is still safe. Before executing a piece of code, a certified verifier ensures that it respects the centralized security policy. Thus, compilers are no longer required to belong to the trusted computing base.

**Acknowledgments**. We thank Luís Caires and the anonymous referees for their comments. Research partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

## References

1. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. *Proc. Fossacs (LNCS 3441)*, 2005.
2. M. Bartoletti, P. Degano, and G. L. Ferrari. Checking risky events is enough for local policies. Proc. 9th ICTCS (LNCS 3701), 2005.
3. M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. *Proc. CSFW*, 2006.
4. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. *Proc. FCS*, 2002.
5. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
6. J. C. Bradfield. On the expressivity of the modal $\mu$-calculus. *Proc. of Int. Symp. on Theoretical Aspects of Computer Science*, 1996.
7. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. *Proc. POPL*, 2000.
8. Ú. Erlingsson and F.B. Schneider. SASI enforcement of security policies: a retrospective. *Proc. of 7th New Security Paradigms Workshop*, 1999.
9. J. Esparza. On the decidability of model checking for several $\mu$-calculi and Petri nets. *Proc. of 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
10. C. Fournet and A.D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
11. Li Gong. Inside Java 2 platform security: architecture, API design, and implementation. Addison-Wesley, 1999.
12. A. Igarashi and N. Kobayashi. Resource usage analysis. *Proc. POPL*, 2002.
13. D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
14. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. *Proc. APLAS (LNCS 3302)*, 2003.
15. G. C. Necula. Proof-carrying code. *Proc. POPL*, 1997.
16. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
17. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.
18. C. Skalka and S. Smith. History effects and verification. *Proc. of APLAS*, 2004.
19. C. Skalka. Trace Effects and Object Orientation. *Proc. PPDP*, 2005.
20. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Proc. 7th IEEE Symposium on Logic in Computer Science*, 1992.
21. P. Thiemann. Enforcing Safety Properties Using Type Specialization *Proc. ESOP*, 2001.
22. D. Walker. A type system for expressive security policies. *Proc. POPL*, 2000.