Drossopoulou and Noble [4] argue persuasively for the need for a means to express policy in object-capability-based systems. We investigate a practical means to realize their aim via the Curry-Howard isomorphism [1]. Specifically, we investigate representing policy as types in a behavioral type system for the RHO-calculus [11], a reflective higher-order variant of the π-calculus [15].

Policy as Types

L.G. Meredith
Biosimilarity, LLC
lgreg.meredith@biosimilarity.com

Mike Stay
Google
stay@google.com

Sophia Drossopoulou
Imperial College, London
s.drossopoulou@imperial.ac.uk

1 Introduction

The object-capability (ocap) security model grew out of the realization that good object-oriented programming practice leads to good security properties. Separation of duties leads to separation of authority; information hiding leads to integrity; message passing leads to authorization; and dependency injection to authority injection. An ocap-secure programming language enforces these patterns: the only way an object can modify any state but its own is by sending messages on the object references it possesses. Authority can then be denied simply by not providing the relevant object reference. Ocap languages do not provide ambient, undeniable authority such as the global variables in JavaScript, static mutable fields in Java and C#, or allowing access to arbitrary objects through pointer arithmetic as in C and C++.

The electronic society is moving steadily towards the object capability model. ECMAScript 5, the standard version of JavaScript adopted by all modern browsers, introduced a new security API that enables the web browser to be turned into an ocap-safe platform; Google's Caja project is an implementation [9]. Brendan Eich, CTO of Mozilla, wrote, "In SpiderMonkey + Gecko in Firefox, and probably in other browsers, we actually use OCap under the hood and have for years. ... Any time we deviate from OCap, we regret it for both security bug and access-checking overhead reasons." [5]

We would like a way to declare the authority that an object ought to possess and then check whether the implementation matches the intent; that is, we would like a language for declaring security policy. Drossopoulou and Noble [4] convincingly argue that none of the current specification methods adequately capture all of the capability policies required to support ocap systems. The most intriguing aspect of the capability policies proposed in [4] is *deniability*. Deniability differs from usual style of specifications in the following two aspects: a) it describes policies which are *open*, i.e. apply to a module as well as all its extensions, and b) describes *necessary* conditions for some effect to take place. For example, a policy may require that in order for any piece of code to modify the balance of a bank account, it needs to have access to that account (access to the account is a necessary condition), and that this property is satisfied by all extensions of the account library (open). In contrast, classical Hoare Logic specifications [6] describe sufficient conditions and are closed. In this paper we concentrate on the use of the RHO-calculus [11] for expressing deniability. To express necessary conditions, we invert the assertion and use bisimulation (e.g. any piece of code which does not have access to the account will be bisimilar to code in which the balance of the account is not modified). To express openness, we adapt the concept of the adjoint to the separation operator, i.e. any further code P which is attached to the current code will be bisimilar to P in parallel with code which does not modify the balance of the account.

In short, we extend the usual interpretation of π-calculus as an ocap language to the RHO-calculus

Submitted to:
BEAT 2014

and demonstrate that the corresponding Hennessy-Milner logic suffices to capture the key notion of deniability. In the larger scheme, this identifies [4]'s notion of policy with the proposition-as-types paradigm, also known as The Curry-Howard isomorphism [1]. The Curry-Howard isomorphism relates formal logic to type theory; it says that propositions are to types as proofs are to programs. Hennessy-Milner logic lets us treat the type of a concurrent process as an assertion that it belongs to a set of processes, all of which satisfy some property. We can treat these properties as contracts governing the behavior of the process; in other words, the language of behavioral types suffices as a security policy language for ocaps.

2 Interpreting capabilities

In [14] we give a compositional interpretation of (a fragment of) Javascript sufficient to encode the central examples of Drossopoulou and Noble's paper in the RHO−*calculus*. Essentially, capabilities are interpreted as channels. In ordinary process calculi channels have no observable internal structure. In the calculus presented here they do, providing a means to reason about collections of channels and hence capabilities.

2.1 The calculus

Before giving the formal presentation of the calculus and logic where we interpret ocap programs and policy, respectively, we begin with a simple example that illustrate a design pattern used over and over in this paper. Here is a mutable single-place cell for storing and retrieving state.

$$
\begin{array}{l}
\mathtt{def}\ \ Cell(slot, state) \Rightarrow\ \{ \\
\quad \mathtt{new}\ \ (v)\ \{ \\
\qquad v!(state) \\
\qquad slot?get(ret) \Rightarrow\ \{\ \ v?(s) \Rightarrow ret!(s) \mid Cell(slot, s)\ \ \}\ \ +\ \ slot?set(s) \Rightarrow\ \{\ \ Cell(slot, s)\ \ \} \\
\quad \} \\
\}
\end{array}
$$

We read this as saying that a *Cell* is parametric in some (initial) *state* and a *slot* for accessing and mutating the cell's state. A *Cell* allocates a private channel *v* where it makes the initially supplied *state* available to its internal computations. If on the channel *slot* it receives a *get* message containing a channel *ret* indicating where to send the state of the cell, it accesses the private channel *v* and sends the value it received on to the *ret* channel; then it resumes behaving as a *Cell*. Alternatively, if it receives a *set* message containing some new state *s*, it simply continues as a *Cell* instantiated with accessor *slot* and state *s*.

Despite the fact that this example bears a striking resemblance to code in any of a number of popular programming libraries, it is only a mildly sugared form of the direct representation in our calculus; and like modern application code, it exhibits encapsulation and separation of implementation from API: a cell's internal access to `state` is kept in a private channel `v` while external access is through `slot`. Likewise this design pattern scales through composition: when translating ocap examples expressed in `ECMAScript` to RHO-calculus we treat the state of an object as a parallel composition of cells comprising its state. The interested reader is invited to see [14] for more examples and details of the translation.

2.1.1 The RHO-calculus

The RHO-calculus [12] is a variant of the asynchronous polyadic π-calculus. When names are polarized [16], the π-calculus is an ocap language. Pierce and Turner [17] defined the programming language Pict as sugar over the polarized π-calculus together with some supporting libraries; unfortunately, the authority provided by the libraries violated the principle of least authority. Košík refactored the libraries to create Tamed Pict, recovering an ocap language, then used it to write a defensively consistent operating system [10]. The RHO-calculus differs from the π-calculus in that names are quoted processes rather than generated by the ν operator. Both freshness and polarization are provided through the use of namespaces at the type level.

We let P, Q, R range over processes and x, y, z range over names, and $\vec{x}$ for sequences of names, $|\vec{x}|$ for the length of $\vec{x}$, and $\{\vec{y}/\vec{x}\}$ as partial maps from names to names that may be uniquely extended to maps from processes to processes [12].

IO	ABSTRACTION	DATA
$M, N ::= 0 \mid x?A \mid x!C \mid M + N$	$A ::= (x_1, \ldots, x_N) \Rightarrow P$	$C ::= (Q_1, \ldots, Q_N)$

PROCESS	NAME
$P, Q ::= M \mid P \mid Q \mid *x$	$x, y ::= @P$

The examples from the previous section use mild (and entirely standard) syntactic sugar: `def` making recursive definitions a little more convenient than their higher-order encodings, `new` making fresh channel allocation a little more convenient and `match` for purely input-guarded choice. The interested reader is directed to [14] for more details.

The calculation of the free names of a process, P, denoted $\mathbf{FN}(P)$ is given recursively by

$$\mathbf{FN}(0) := \emptyset \qquad \mathbf{FN}(x?(y_1, \ldots, y_N) \Rightarrow P) := \{x\} \cup (\mathbf{FN}(P) \setminus \{y_1, \ldots y_N\})$$

$$\mathbf{FN}(x!(Q_1, \ldots, Q_N)) := \{x\} \cup \bigcup \mathbf{FN}(Q_i) \qquad \mathbf{FN}(P \mid Q) := \mathbf{FN}(P) \cup \mathbf{FN}(Q) \qquad \mathbf{FN}(*x) := \{x\}$$

An occurrence of x in a process P is *bound* if it is not free.

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, that satisfies the following laws:

$$P \mid 0 \equiv P \equiv 0 \mid P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

As discussed in [11] the calculus uses an equality, $\equiv_N$, pronounced name-equivalence, recursively related to α-equivalence and structural equivalence, to judge when two names are equivalent, when deciding synchronization and substitution.

The engine of computation in this calculus is the interaction between synchronization and a form of substitution, called semantic substitution. Semantic substitution differs from ordinary syntactic substitution in its application to a dropped name. For more details see [12]

$$(*x)\{\widehat{@Q/@P}\} \quad := \quad \begin{cases} Q & x \equiv_N @P \\ *x & otherwise \end{cases}$$

Finally equipped with these standard features we can present the dynamics of the calculus.

The reduction rules for RHO-calculus are

COMM
$$\frac{x_0 \equiv_N x_1, |\vec{y}| = |\vec{Q}|}{x_0?(\vec{y}) \Rightarrow P \mid x_1!(\vec{Q}) \rightarrow P\{@\vec{Q}/\vec{y}\}}$$

PAR
$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

EQUIV
$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

The context rules, PAR and EQUIV, are entirely standard and we do not say much about them here. The communication rule makes it possible for agents to synchronize at name-equivalent guards and communicate processes packaged as names. Here is a simple example of the use of synchronization, reduction and quasiquote: $x(z) \Rightarrow w!(y!(*z)) \mid x!(P) \rightarrow w!(y!(P))$. The input-guarded process, $x(z) \Rightarrow w!(y!(*z))$, waits to receive the code of P from the output-guarded data, $x!(P)$, and then reduces to a new process the body of which is rewritten to include P, $w!(y!(P))$.

Bisimulation, one of the central features of process calculi provides an effective notion of substitutability together with a range of powerful proof techniques. For this paper we focus principally on the use of logic to interpret policy and thus need connect the ideas of logical equivalence and bisimulation. The main theorem of the next section relates the logic to bisimulation. Meanwhile, the experienced reader should note that the set of names of the RHO-calculus is *global*. It is impossible, in the grammar of processes, to guard terms from being placed into contexts that can potentially observe communication, so we provide a place for reasoning about such limitations on the scope of observation in the theory of bisimulation.

Definition 2.1.1 *An* observation relation*, $\downarrow_{\mathcal{N}}$, over a set of names, $\mathcal{N}$, is the smallest relation satisfying the rules below.*

OUT-BARB
$$\frac{y \in \mathcal{N},\ x \equiv_N y}{x!(*v) \downarrow_{\mathcal{N}} x}$$

PAR-BARB
$$\frac{P \downarrow_{\mathcal{N}} x \text{ or } Q \downarrow_{\mathcal{N}} x}{P \mid Q \downarrow_{\mathcal{N}} x}$$

We write $P \Downarrow_{\mathcal{N}} x$ if there is Q such that $P \Rightarrow Q$ and $Q \downarrow_{\mathcal{N}} x$.

Definition 2.1.2 *An* $\mathcal{N}$-barbed bisimulation *over a set of names, $\mathcal{N}$, is a symmetric binary relation $\mathcal{S}_{\mathcal{N}}$ between agents such that $P\ \mathcal{S}_{\mathcal{N}}\ Q$ implies:*

1. *If $P \rightarrow P'$ then $Q \Rightarrow Q'$ and $P'\ \mathcal{S}_{\mathcal{N}}\ Q'$.*
2. *If $P \downarrow_{\mathcal{N}} x$, then $Q \Downarrow_{\mathcal{N}} x$.*

P is $\mathcal{N}$-barbed bisimilar to Q, written $P \dot{\approx}_{\mathcal{N}} Q$, if $P\ \mathcal{S}_{\mathcal{N}}\ Q$ for some $\mathcal{N}$-barbed bisimulation $\mathcal{S}_{\mathcal{N}}$.

3 Interpreting policy

In this section we present enough of namespace logic to interpret deniability policy specification. For the reader familiar with logics of concurrency the core idea is to use rely-guarantee combined with the adjunct to spatial separation. Pragmatically, a visual inspection of the code for *Cell* reveals that if a process *P*'s names are fresh with respect to *slot*, then *P* cannot directly affect any $Cell(slot, state)$. We can use the logic presented below to state and verify the fact that *P* cannot affect any `Cell` and then generalize and scale the approach to wider settings.

3.1 Namespace logic

Namespace logic resides in the subfamily of Hennessy-Milner logics known as spatial logics, discovered by Caires and Cardelli [2]. Thus, in addition to the action modalities, we also find at the logical level formulae for *separation* corresponding to the structural content of the parallel operator at the level of the calculus. Likewise, we have quantification over names.

There are important differences between the logic presented here and Caires' logic. Because it is built on a reflective programming language (the RHO-calculus) it also has formulae, $\ulcorner\phi\urcorner$, for describing classes of names. Said another way, the inhabitants of types of the form $\ulcorner\phi\urcorner$ are names and hence these types have the right to be called namespaces [11]. The next two examples show how useful namespaces are in the security setting.

Suppose that $\ulcorner\phi\urcorner$ describes some namespace, *i.e.* some collection of names. We can insist that a process restrict its next input to names in that namespace by insisting that it witness the formula: $\langle\ulcorner\phi\urcorner?b\rangle\, true \,\&\neg\langle\ulcorner\neg\phi\urcorner?b\rangle true$ which simply says the the process is currently able to take input from a name in the namespace $\ulcorner\phi\urcorner$ and is not capable of input on any name not in that namespace.

In a similar manner, we can limit a server to serving only inputs in $\ulcorner\phi\urcorner$ throughout the lifetime of its behavior with $\mathsf{rec}\, X\,.\,\langle\ulcorner\phi\urcorner?b\rangle X\&\neg\langle\ulcorner\neg\phi\urcorner?b\rangle X$ [1] This formula is reminiscent of the functionality of a firewall, except that it is a *static* check. A process witnessing this formula will behave as though it were behind a firewall admitting only access to the ports in $\ulcorner\phi\urcorner$ without the need for the additional overhead of the watchdog machinery.

We will use refinements this basic technique of walling off a behavior behind a namespace when reasoning about the ocap examples and especially in the context of deniability.

3.1.1 Logical syntax and semantics

SET	MONOID	BEHAVIOR	CLOSURE
$\phi,\psi ::= true \mid \neg\phi \mid \phi\&\psi$	$\mid 0 \mid \phi \mid \psi$	$\mid \urcorner b\ulcorner \mid a!(\phi) \mid \langle a?b\rangle\phi$	$\mid \forall n:\psi\,.\,\phi \mid \mathsf{rec}\, X\,.\,\phi \mid X$

NAMESPACE	NAME
$a ::= \ulcorner\phi\urcorner \mid b$	$b ::= \ulcorner P\urcorner \mid n$

In the examples below, we freely employ the usual DeMorgan-based syntactic sugar. For example,

[1] Of course, this formula also says the server never goes down, either—or at least is always willing to take such input.

$\phi \Rightarrow \psi := \neg(\phi \& \neg\psi)$ and $\phi \vee \psi := \neg(\neg\phi \& \neg\psi)$. Also, when quantification ranges over all of $@Proc$, as in $\forall n : \ulcorner true \urcorner . \phi$, we omit the typing for the quantification variable, writing $\forall n . \phi$. Further, we let $PForm$ denote the set of formulae generated by the ϕ-production, $QForm$ denote the set of formulae generated by the a-production and $\mathscr{V}$ denote the set of propositional variables used in the rec production. Additionally, we let $PForm_{-Par}$ denote the separation-free fragment of formulae.

Inspired by Caires' presentation of spatial logic [2], we give the semantics in terms of sets of processes (and names). We need the notion of a valuation $v : \mathscr{V} \to \wp(Proc)$, and use the notation $v\{\mathscr{S}/X\}$ to mean

$$v\{\mathscr{S}/X\}(Y) \quad = \quad \begin{cases} S & Y = X \\ v(Y) & otherwise \end{cases}$$

The meaning of formulae is given in terms of two mutually recursive functions,

$$[\![-]\!](-) : PForm \times [\mathscr{V} \to \wp(Proc)] \to \wp(Proc)$$
$$((-))(-) : QForm \times [\mathscr{V} \to \wp(Proc)] \to \wp(@Proc)$$

taking a formula of the appropriate type and a valuation, and returning a set of processes or a set of names, respectively.

$$[\![true]\!](v) := Proc \qquad [\![\neg\phi]\!](v) := Proc / [\![\phi]\!](v) \qquad [\![\phi \& \psi]\!](v) := [\![\phi]\!](v) \cap [\![\psi]\!](v)$$

$$[\![0]\!](v) := \{P : P \equiv 0\} \qquad [\![\phi \mid \psi]\!](v) := \{P : \exists P_0, P_1 . P \equiv P_0 \mid P_1,\ P_0 \in [\![\phi]\!](v),\ P_1 \in [\![\psi]\!](v)\}$$

$$[\![\ulcorner b \urcorner]\!](v) := \{P : \exists x . P \equiv *x,\ x \in ((b))(v)\}$$

$$[\![a!(\phi)]\!](v) := \{P : \exists P' . P \equiv x!(P'),\ x \in ((a))(v),\ P' \in [\![\phi]\!](v)\}$$

$$[\![\langle a?b \rangle \phi]\!](v) := \{P : \exists P' . P \equiv x?(y) \Rightarrow P', x \in ((a))(v),\ \forall c . \exists z . P'\{z/y\} \in [\![\phi\{c/b\}]\!](v)\}$$

$$[\![\forall n : \psi . \phi]\!](v) := \cap_{x \in ((@\psi))(v)} [\![\phi\{x/n\}]\!](v) \qquad [\![\mathsf{rec}\, X . \phi]\!](v) := \cup\{\mathscr{S} \subseteq Proc : \mathscr{S} \subseteq [\![\phi]\!](v\{\mathscr{S}/X\})\}$$

$$((\ulcorner \phi \urcorner))(v) := \{x : x \equiv_N @P, P \in [\![\phi]\!](v)\} \qquad ((\ulcorner P \urcorner))(v) := \{x : x \equiv_N @P\}$$

We say P witnesses ϕ (resp., x witnesses $\ulcorner \phi \urcorner$), written $P \models \phi$ (resp., $x \models \ulcorner \phi \urcorner$) just when $\forall v . P \in [\![\phi]\!](v)$ (resp., $\forall v . x \in [\![\ulcorner \phi \urcorner]\!](v)$).

Theorem 3.1.1 (Equivalence) $P \dot{\approx} Q \Leftrightarrow \forall \phi \in PForm_{-Par} . P \models \phi \Leftrightarrow Q \models \phi$.

A consequence of this theorem is that there is no algorithm guaranteeing that a check for the witness relation will terminate. However there is a large class of interesting formulae for which the witness check does terminate, see [2]. *Nota bene*: including separation makes the logic strictly more observant than bismulation.

3.2 Applications to policy

Adapting the firewall formula, let $soleAccess(slot) := \mathsf{rec}\, X\,.\, \langle slot?b\rangle X \& \neg\langle \neg slot?b\rangle X$, and $noAccess(slot) := \mathsf{rec}\, X\,.\, \neg\langle slot?b\rangle X$; then we have $Cell(slot,state) \models soleAccess(slot)$ which implies $\mathtt{new}(slot)\{Cell(slot,state)\} \dot{\approx} 0$. If we have that $P \models noAccess(slot)$, then $\forall\, state.\mathtt{new}(slot)\{P \mid Cell(slot,state)\} \dot{\approx} P$. For, by hypothesis, *slot* cannot be free in P, thus $\mathtt{new}(slot)\{P \mid Cell(slot,state)\} \equiv P \mid \mathtt{new}(slot)\{Cell(slot,state)\}$ which means $P \mid \mathtt{new}(slot)\{Cell(slot,state)\} \dot{\approx} P \mid 0 \equiv P$. More generally, note that objects involve not just one cell, but many. By restricting those cells to slots that live in namespace $Slot := \ulcorner \phi \urcorner$ for some ϕ we can scale our formula to separate objects from a given environment.

Notice the shape of the logical statement above is: if $Q \models \phi$ then when $P \models \psi$ we have that $\mathtt{new}(x_1,\ldots,x_N)\{P \mid Q\} \dot{\approx} S$ for some properties ϕ and ψ and some behavior, S. In light of theorem 3.1.1 we can, in principle, transform this into a statement of the form, "If $Q \models \phi$, then when $P \models \psi$ we have that $\mathtt{new}(x_1,\ldots,x_N)P \mid Q \models \omega$ for some properties ϕ, ψ, and ω." This is the shape of our encoding of deniability policies. The key insight here is to mediate between closed-world and open-world forms of deniability. In this open-world formulation we do not quantify over all potential contexts and attackers, but work just with those that can be relied upon to use the API, even in attacks. Further, this encoding emphasizes necessary rather than sufficient conditions. Finally, notice that the form of the statement remains true to the intent of deniability even if the logic is *more* discriminating than bisimulation—which is the case if we include the separation operator (as noted in [2]). In the presence of a logic with separation this shape turns out to be a form of rely-guarantee identified by Honda in [7] and, as Honda observes, can be internalized in the logic as the adjunct to separation. Thus, we include a form of rely-guarantee formulae, $\phi \rhd_{\vec{x}} \psi$, where $\vec{x} = \{x_1,\ldots,x_N\}$, with semantics $[\![\phi \rhd_{\vec{x}} \psi]\!](v) = \{P : \forall Q.Q \in [\![\phi]\!](v) \Rightarrow \mathtt{new}(x_1,\ldots,x_N)\{P \mid Q\} \in [\![\psi]\!](v)\}$, as the logical form that mediates between closed-world and open-world formulations of deniability.

4 Conclusions and future work

We take the simplicity and flexibility of separating cells (and compositions of cells) from environments as evidence that the approach is natural and commensurate with the expressiveness demands of policy in an ocap setting. We are also interested in what this interpretation says specifically about deniability style policies. In this connection it is important to note that a lot is already known about the complexity characteristics of spatial-behavioral logics. An interpretation of deniability via rely-guarantee suggests suggests to is a very powerful policy construct for as [3] note the addition of the adjunct to separation is not a conservative extension!

For the interested reader, source code for implementations of an interpreter for the RHO-calculus and a model-checker for namespace logic are available upon request and a more industrial scale version of the core concepts are used heavily in the `SpecialK` library available on `github` [13]. This library is already in use in industrial projects such as the Protunity Business Matching Network [8].

Acknowledgments. The authors wish to thank Mark Miller for thoughtful and stimulating conversation about policy, object capabilities and types.

References

[1] Samson Abramsky, *Proofs as processes*, Theor. Comput. Sci. **135** (1992), no. 1, 5–9.

[2] Luís Caires, *Behavioral and spatial observations in a logic for the pi-calculus.*, FoSSaCS, 2004, pp. 72–89.

[3] Luís Caires and Etienne Lozes, *Elimination of quantifiers and undecidability in spatial logics for concurrency*, 2004.

[4] Sophia Drossopoulou and James Noble, *The need for capability policies*, Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs (New York, NY, USA), FTfJP '13, ACM, 2013, pp. 6:1–6:7.

[5] Brendan Eich, `https://mail.mozilla.org/pipermail/es-discuss/2013-March/029080.html`, 2013.

[6] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580.

[7] Kohei Honda and Nobuko Yoshida, *A unified theory of program logics: an approach based on the π-calculus*, Proceedings of the 2008 international conference on Visions of Computer Science: BCS International Academic Conference (Swinton, UK, UK), VoCS'08, British Computer Society, 2008, pp. 259–274.

[8] Wadood Ibrahim, Jassen Klassen, and Lucius G. Meredith, *Protunity business matching network*, `http://www.protunity.com`, Launched April 2013.

[9] Google Inc., *Html service: Caja sanitization*, `https://developers.google.com/apps-script/guides/html-service-caja`, 2013.

[10] Matej Košík, *A Contribution to Techniques for Building Dependable Operating Systems*, Ph.D. thesis, Slovak University of Technology in Bratislava, May 2011.

[11] L. Gregory Meredith and Matthias Radestock, *Namespace logic: A logic for a reflective higher-order calculus.*, in *TGC* [12], pp. 353–369.

[12] ———, *A reflective higher-order calculus.*, Electr. Notes Theor. Comput. Sci. **141** (2005), no. 5, 49–67.

[13] Lucius G. Meredith, *Specialk scala library*, `https://github.com/leithaus/SpecialK`, Launched April 2013.

[14] Lucius G. Meredith, Mike Stay, and Sophia Drossopoulou, *Policy as types*, CoRR **abs/1307.7766** (2013), 1–10.

[15] Robin Milner, *The polyadic π-calculus: A tutorial*, Logic and Algebra of Specification **Springer-Verlag** (1993), 1–34.

[16] Martin Odersky, *Polarized name passing.*, FSTTCS (P. S. Thiagarajan, ed.), Lecture Notes in Computer Science, vol. 1026, Springer, 1995, pp. 324–337.

[17] Benjamin C. Pierce and David N. Turner, *Pict: A programming language based on the pi-calculus*, Proof, Language and Interaction: Essays in Honour of Robin Milner (Gordon Plotkin, Colin Stirling, and Mads Tofte, eds.), MIT Press, 2000, pp. 455–494.