Int J Inf Secur (2004) 00: 1–18 / Digital Object Identifier (DOI) 10.1007/s10207-004-0049-5

Static use-based object confinement

Christian Skalka[1], **Scott Smith**[2]

[1] The University of Vermont, Burlington, VT 05405, USA e-mail: skalka@cs.uvm.edu
[2] The Johns Hopkins University, Baltimore, MD 21218, USA e-mail: scott@cs.jhu.edu

Published online: ■■ 2004 – © Springer-Verlag 2004

Abstract. The confinement of object references is a significant security concern for modern programming languages. We define a language that serves as a uniform model for a variety of confined object reference systems. A *use-based* approach to confinement is adopted, which we argue is more expressive than previous communication-based approaches. We then develop a readable, expressive type system for static analysis of the language, along with a type safety result demonstrating that run-time checks can be eliminated. The language and type system thus serve as a reliable, declarative, and efficient foundation for secure capability-based programming and object confinement.

Keywords: Programming languages – Security – Confinement – Type systems

1 Introduction

The confinement of object references is a significant security concern in languages such as Java. Aliasing and other features of OO languages can make this a difficult task; recent work [1, 5, 8, 9, 29] has focused on the development of type systems for enforcing various containment policies in the presence of these features. This paper describes a new language and type system that implements an object confinement model that is more general than previous systems and that is based on a different notion of security enforcement.

Object confinement is closely related to *capability*-based security, utilized in several operating systems such as EROS [24] and also in programming language (PL) architectures such as J-Kernel [12], E [10], and Secure Network Objects [28]. A capability can be defined as a reference to a data segment, along with a set of access rights to the segment [14]. An important property of capabilities is that they are *unforgeable*: they cannot be faked or reconstructed from partial information. In Java, object references are likewise unforgeable, a property enforced by the type system; thus, Java can be considered a statically enforced capability system.

So-called *pure* capability systems rely on their high-level design for safety, without any additional system-level mechanisms for enforcing security. Other systems *harden* the pure model by layering other mechanisms over pure capabilities to provide stronger system-level enforcement of security; the `private` and `protected` modifiers in Java are examples of this. Types improve the hardening mechanisms of capability systems by providing a declarative statement of security policies, as well as improving run-time efficiency through static, rather than dynamic, enforcement of security. Our language model and static type analysis focuses on capability hardening, with enough generality to be applicable to a variety of systems, and serves as a foundation for studying object protection in OO languages.

Our approach is similar to [5], which establishes theoretical foundations for statically enforcing a fixed set of access rights on capabilities at the programming language level by proving a type safety result that demonstrates that run-time access checks are unnecessary in an idealized semantics. However, our notion of access rights is much more general, being applicable to any object method, rather than just a fixed set, and can be modulated so that different access rights may be assigned to the same capability in different domains of usage. In this respect our approach is similar to the access control system for process calculi presented in [13], where location names embody capabilities for mobile agents, and known channels at those locations may be modulated for varying degrees of resource authorization. As in [5], we adopt the traditional spirit of access control for capabilities, which

essentially comprises a lightweight access check of the interface associated with a capability at the point of usage. This distinguishes the object access control in this setting from the resource access control mechanisms available with, e.g., security automata [30], or stack-inspection-style access control [11, 20].

Since our work is intended as the foundation for a distinct *use-based* view of access control (Sect. 2.1), we focus on the theoretical underpinnings of a relevant static analysis. Thus, we do not treat more advanced language features such as inheritance and concurrency, but rather a core object-based calculus. Also, while the literature on object containment has included analyses for global properties such as disjointness of effects [8], noninterference [3], and safety guarantees in the presence of modularity [1, 2, 15, 16], we concentrate on the enforcement of the system of local checks that are basic components of the language. In short, our aim is to study, from first principles, static object confinement from a general use-based perspective – to characterize this perspective via formalization, to establish the basis for a relevant, practical type system by defining the type language, proof, and inference systems, and to prove these systems sound.

Outline of the presentation. The high-level organization of this paper is as follows. In Sect. 2 we describe the fundamental ideas and basic usage of our language, called pop. In Sect. 3 we give the formal syntactic and semantic definition of pop. In Sect. 4 we show how pop can be simulated, via transformation of terms, in another language. This transformation serves as the foundation for the development of a type discipline for pop in Sect. 5. In Sect. 6 we give some detailed examples of programming in pop that highlight the usefulness of its type discipline. We conclude with some remarks about related and future work in Sect. 7.

2 Overview of the pop system

In this section we informally describe some of the ideas and features of our language, called pop (an acronym for "programming with object protection") and show how they improve upon previous systems. As will be demonstrated in Sect. 6, pop is sufficient to implement various OO language features, e.g., classes with methods and instance variables, but with stricter and more reliable security.

2.1 Use-based vs. communication-based security

Our approach to object confinement is related to previous work on containment mechanisms for object-oriented languages [4, 9, 29], but with a different basis. These previous containment mechanisms rely on a *communication*-based approach to security; some form of barriers between objects, or domain boundaries, are specified, and security is concerned with communication of objects (or object references) across those boundaries. In our *use*-based approach, we also specify domain boundaries, but security is concerned with how objects are *used* within these boundaries. Practically speaking, this means that security checks on an object are performed when it is used (selected) rather than communicated. A similar use-based perspective has been exploited previously in a process calculi setting [13], where types specify the particular channels available for authorized use at a known location.

The main advantage of the use-based approach is that security specifications may be more fine-grained; in a communication-based approach, we are restricted to a whole-object "what-goes-where" security model, while with a use-based approach we may be more precise in specifying what methods of an object may be used within various domains. This is particularly relevant to access control. Use-based security also more closely corresponds to traditional capability-based security models in practice, where capabilities are not just references but are references plus an interface specifying access rights.

In addition, our use-based security model allows "tunneling" of objects: a capability may pass through a domain where its use is disallowed, provided it is not used in that domain. This supports the multitude of protocols that rely on an intermediary that is not fully trusted. In a communication-based model capabilities are prevented from passing through unauthorized domains, so tunneling is impossible.

2.2 Static protection domains

The pop language is an object-based calculus, where objects are defined as collections of method definitions. For example, substituting the notation ... for the syntactic details, the definition of a file object with read and write methods would appear as follows:

$$[\mathrm{read}() = \ldots\,, \mathrm{write}(x) = \ldots] \cdot \ldots \cdot \ldots$$

Additionally, every object definition statically asserts membership in a specific *protection domain* d, so that expanding on the above we could have:

$$[\mathrm{read}() = \ldots\,, \mathrm{write}(x) = \ldots] \cdot d \cdot \ldots$$

While the system requires that all objects be annotated with a domain, the *meaning* of these domains is flexible and open to interpretation. Our system, considered in a pure form, is a core analysis that may be specialized for particular applications. For example, domains may be interpreted as code owners, or they may be interpreted as denoting regions of static scope, e.g., package scope.

Along with domain labels, the language provides a method for specifying a security policy, dictating how domains may interact, via *user interface* definitions φ. Each object is annotated with a user interface, so that,

letting φ be an appropriately defined user interface and again expanding on the above, we could have:

$$[\text{read}() = \ldots\,, \text{write}(x) = \ldots] \cdot d \cdot \varphi\,.$$

We describe user interfaces more precisely below and illustrate and discuss relevant examples in Sect. 6. For now, we note that the flexibility in the interpretation of domains implies a flexibility in the style of policies that may be enforced: e.g., if domains are interpreted as code-owner labels, then the policy is access control, while if domains are interpreted as static scope, then the policy is a use-based access modifier mechanism.

2.3 Object interfaces

Other secure capability-based language systems have been developed [10, 12, 28] that include a notion of an access-rights interface, in the form of object types. Our system provides a more fine-grained mechanism: for any given object, its user-interface definition φ may be defined so that different domains are given more or less restrictive views of the same object, and these views are statically enforced. Note that the use-based, rather than communication-based, approach to security is an advantage here, since the former allows us to more precisely modulate *how* an object may be used by different domains, via object method interfaces.

For example, we can imagine that any object in domain d is a "friend" and should be given free reign over other objects in d, whereas objects in domain d' are somewhat trusted but potentially hostile, so that we might wish such objects to read data in d but not be able to alter it. Thus, returning to our previous example, an appropriate definition of φ in the file object definition, given these security preconceptions, would be:

$$\{d \mapsto \{\text{read}, \text{write}\}\,, d' \mapsto \{\text{read}\}\}\ .$$

User interfaces may additionally contain mappings for a *default* user ∂, which allows the programmer to specify interfaces for domains that may not be known at compile time. Thus, the system allows for a degree of "open-endedness" in its design. Again returning to the previous example, if our policy was to allow *any* domain read access to files in domain d, we could define φ as:

$$\{d \mapsto \{\text{read}, \text{write}\}\,, \partial \mapsto \{\text{read}\}\}\ .$$

The notation ∂ matches any domain. As is the case for normal interface specifications, the access rights associated with default interfaces are statically enforced.

The user interface is a mapping from domains to access rights – that is, to sets of methods in the associated object that each domain is authorized to use. This looks something like an ACL-based security model; however, ACLs are defined to map *principals* to privileges. Domains, on the other hand, are fixed boundaries in the code that may have nothing to do with principals. The practical usefulness of a mechanism with this sort of flexibility has been described in [6], in application to mobile programs. Other applications and more detailed examples are discussed in Sect. 6, including an encoding of `private` and `protected` method and instance variable modifiers.

2.4 Casting

We also provide a *casting* mechanism that allows removal of access rights from particular views of an object, allowing a greater attenuation of security when necessary. Again, this casting discipline is statically enforced. For example, letting o be the pop object defined immediately above, if some circumstance suggests that we should no longer allow objects in domain d' read access to files in d, then we may make the following cast:

$$o_!(d', \varnothing)\,.$$

This removes all of d' access privileges on o by setting the set of d''s accessible methods to $\varnothing$. Significantly, we allow only "upcasts", so that privileges can be removed but never added.

Type systems already have a built-in notion of interface and of restriction of interfaces, via subtyping and subsumption. Our system is inspired by and sits on a foundation of subtyping, but it is significantly more general. Most importantly, privileges can be restricted by subtyping in standard systems, but this is only with respect to two implicit domains: the local one and everything else. With our explicit domains and fine-grained user interface definitions, casting restrictions may be significantly more fine-grained.

2.5 Weakening

Capability weakening [24] is a sort of "deep-casting" casting mechanism that applies a cast to an object *and* any objects subsequently returned by that object. It can be used to enforce, e.g., "transitive read-only" security properties in confined systems [24]. We generalize weak capabilities in the system presented here and statically enforce weakening properties via types. For example, if o is a directory object and delete is a directory object method that allows deletion of entities in a directory, then the expression

$$\mathbf{weak}_{\{\text{delete}\}}(o)$$

denotes a weakening of o such that deletion within that directory is disallowed, and furthermore, if $m \neq \text{delete}$ is an accessible method of o that returns another directory object o', then o' will be similarly weakened to disallow deletion. The type system statically enforces this mechanism in a flexible manner.

2.6 Rights amplification

Capability-based security systems support various forms of rights amplification, which is the temporary and disciplined amplification of rights in certain program contexts. One such form is effected by indirection. For example, letting o be the file object as defined immediately above, and recalling that d was the only domain allowed write access to o, we may allow another object in domain d to function as a write-access "proxy" to o, as in the following definition:

$$[\text{proxywrite}(x) = o.\text{write}(x)] \cdot d \cdot \{\partial \mapsto \{\text{proxywrite}\}\} \ .$$

Any object in any domain may use this object to gain write-access to o, though *direct* write-access to o is restricted. This example is extreme and not a recommended programming style, but a limited use "by proxy" of capabilities not directly held is a common idiom in capability-based programming. This must also be kept in mind when a capability is doled out – the doler must be aware of both direct and indirect actions allowed by it.

3 The pop language definition

We now formally define the syntax and operational semantics of pop, an object-based language with state and capability-based security features, described informally in the previous sections.

3.1 Syntax

The grammar for pop is defined in Fig. 1. It includes a countably infinite set of identifiers $\mathcal{D}$ that we refer to as *protection domains*. The definition also includes the following notation for method lists ϱ:

$$(m_i(x) = e_i \ ^{0<i\leq n}) \triangleq (m_1(x) = e_1, \ldots, m_n(x) = e_n) \, .$$

Henceforth we will use a similar vector abbreviation notation for all language forms, with obvious meaning. We write $(m(x) = e) \in \varrho$ iff ϱ is of the form $(\ldots, m(x) = e, \ldots)$. Read-write cells are defined as primitives, with a cell constructor $\text{ref}_\varphi v$ that generates a read-write cell containing v, with user interface φ. The object-weakening mechanism described in the previous section is included as $\mathbf{weak}_\iota(o)$.

Object definitions are of the form $[\varrho] \cdot d \cdot \varphi\backslash_\iota$, where ι carries any methods removed by weakening. For convenience, and to retrieve the notation presented in the previous section, we define the syntactic sugar $(co \cdot \varphi) \triangleq (co \cdot \varphi\backslash_\varnothing)$. Self objects $[\varrho]$ are run-time entities, the dynamic implementation of self, and are disallowed in top-level programs.

User interfaces φ are *total* mappings from domain identifiers to sets of method names. Since they are user defined in programs, the following syntactic sugar is provided, allowing a finite specification of interfaces by implicitly mapping unspecified domains to $\varnothing$:

$$\left\{d_i \mapsto \iota_i \ ^{0<i\leq n}\right\} \triangleq \left\{d_i \mapsto \iota_i \ ^{0<i\leq n}, d_{i+1} \mapsto \varnothing, \ldots\right\} \, .$$

We require that, for any φ and d, the method names $\varphi(d)$ be a subset of the method names in the associated object. Note that object method definitions may contain the distinguished identifier $\mathbf{s}$, which denotes *self* and which is bound in object definitions; objects always have full access to themselves via the identifier $\mathbf{s}$. We require that self never appear "bare" – that is, the variable $\mathbf{s}$ must always appear in the context of a method selection $\mathbf{s}.m(e)$. This restriction ensures that $\mathbf{s}$ cannot escape its own scope, unintentionally providing a "back door" to the object. Self, and associated semantics, is discussed more thoroughly below.

For the purposes of bookkeeping in the stack-based semantics presented in the next section, we also have *framed* expressions $\cdot e \cdot$, denoting a region of code associated with a particular stack frame. Objects with framed subexpres-

$m, x, \mathbf{s}, \text{set}, \text{get} \in ID, \ \iota \subseteq ID$		*identifiers*
$l \in Loc$		*locations*
$d \in \mathcal{D}, \ D \subseteq \mathcal{D}$		*domains*
$\varphi \in \mathcal{D} \to 2^{ID}$		*interfaces*
ϱ	$::= m_i(x) = e_i \ ^{0<i\leq n}$	*method lists*
s	$::= [\varrho]$	*self objects*
co	$::= [\varrho] \cdot d \mid l$	*core objects*
o	$::= co \cdot \varphi\backslash_\iota \mid s$	*objects*
v	$::= x \mid o$	*values*
e	$::= v \mid e.m(e) \mid e_!(d, \iota) \mid \mathsf{let}\, x = v\, \mathsf{in}\, e \mid \text{ref}_\varphi e \mid \mathbf{weak}_\iota(e) \mid \cdot e \cdot$	*expressions*
E	$::= [\,] \mid E.m(e) \mid v.m(E) \mid E_!(d, \iota) \mid \text{ref}_\varphi E \mid \mathbf{weak}_\iota(E) \mid \cdot E \cdot$	*evaluation contexts*

Fig. 1. Grammar for pop

sions are disallowed. We define some convenient language for discussing framed expressions.

Definition 1. *The* frame depth *of an evaluation context is inductively defined as follows: the frame depth of* $[\,]$ *is* 0*, the frame depth of* $\cdot E\cdot$ *is* 1 *plus the frame depth of* E*, and the frame depth of any other context form* E *is the frame depth of* E*'s subcontext.*

3.2 Operational semantics

The small-step operational semantics for pop is defined in Fig. 2 as the relation $\rightarrow$ on *configurations* δ, e, σ, where *stores* σ are partial mappings from locations l to values v, and δ is a nonempty *domain stack*, the top element of which is called the *current domain*, representing the resident domain of the current activation. The use of a stack is necessary for bookkeeping during nested calls; for example, if objects o and o' are assigned to domains d and d' respectively and $o.m$ calls $o'.m'$ during execution, while $o'.m'$ executes, the current domain will be d', but when $o'.m'$ terminates and returns control to $o.m$, d must be retrieved as the current domain. Notation and language relevant to domain stacks in this presentation are defined follows:

Definition 2. *Domain stacks are inductively defined as:*

$\delta ::= nil \mid d :: \delta$ *domain stacks*.

The length *of a domain stack* $(d_1 :: \cdots :: d_n :: nil)$ *is* n*. The domain stack reversal function* rev *is defined as:*

$$\mathrm{rev}(d_1 :: \cdots :: d_n :: nil) \triangleq (d_n :: \cdots :: d_1 :: nil)\,.$$

The notation $f[x \mapsto v]$ denotes the function that maps x to v and otherwise is equivalent to f. If $x \notin \mathrm{dom}(f)$, $f[x \mapsto v]$ denotes the function that extends f, mapping x to v. We define $\varphi(d, d') \triangleq \varphi(d) \cup \varphi(d')$. Substitution is defined as one may expect, with the following caveat:

Definition 3. *The "self" identifier* $\mathbf{s}$ *is bound by object definitions, so in particular* $o[[\varrho]/\mathbf{s}] = o$*; otherwise, substitution is defined as usual.*

Additionally, we require that $\rightarrow$ be defined only on *well-formed* configurations, which we elaborate as follows:

Definition 4. *A configuration* $d :: \delta, e, \sigma$ *is* well-formed *iff* e *is closed and there exists* E *and unframed* e' *such that* $e = E[e']$ *and the frame depth of* E *equals the length of* δ*.*

Corollary 1. *If* $d :: \delta, e, \sigma$ *is well-formed and* $e = E[e']$ *with* e' *unframed, then the frame depth of* E *equals the length of* δ*.*

Hereafter we consider only well-formed configurations. It is easy to see that these well-formedness requirements are sensible via the following lemma, the proof of which follows by a straightforward case analysis:

Lemma 1. *If a well-formed configuration* $d :: \delta, e, \sigma$ *is stuck, then* $e = E[e']$*, where* e' *is of the following form:*

1. $(co \cdot \varphi\backslash_\iota).m(v)$*, where* $m \in \iota$ *or* $m \notin \varphi(d, \partial)$
2. $[\varrho].m(v)$ *and* $(m(x) = e) \notin \varrho$
3. $(l \cdot \varphi\backslash_\iota).m(v)$ *and* $m \notin \{set, get\}$
4. $(co \cdot \varphi\backslash_\iota)\,!\,(d', \iota')$ *where* $\iota' \not\subseteq \varphi(d', \partial)$

$$d :: \delta, ([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v), \sigma \hookrightarrow d' :: d :: \delta, \cdot(\mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x]))\cdot, \sigma \quad (send)$$
$$\text{if } (m(x) = e) \in \varrho \text{ and } m \in (\varphi(d, \partial)\backslash\iota)$$

$$\delta, [\varrho].m(v), \sigma \hookrightarrow \delta, e[[\varrho]/\mathbf{s}][v/x], \sigma \quad (self)$$
$$\text{if } (m(x) = e) \in \varrho$$

$$\delta, (co \cdot \varphi\backslash_\iota)\,!\,(d', \iota'), \sigma \hookrightarrow \delta, (co \cdot (\varphi[d' \mapsto \iota'])\backslash_\iota), \sigma \quad (cast)$$
$$\text{if } \iota' \subseteq \varphi(d', \partial)$$

$$\delta, \mathbf{weak}_\iota(co \cdot \varphi\backslash_{\iota'}), \sigma \hookrightarrow \delta, co \cdot \varphi\backslash_{(\iota \cup \iota')}, \sigma \quad (weaken)$$

$$\delta, \mathrm{ref}_\varphi v, \sigma \hookrightarrow \delta, l \cdot \varphi\backslash_\varnothing, \sigma[l \mapsto v] \quad (newcell)$$
$$\text{if } l \notin \mathrm{dom}(\sigma)$$

$$d :: \delta, (l \cdot \varphi\backslash_\iota).\mathrm{set}(v), \sigma \hookrightarrow d :: \delta, \mathbf{weak}_\iota(v), \sigma[l \mapsto v] \quad (set)$$
$$\text{if } l \in \mathrm{dom}(\sigma) \text{ and } \mathrm{set} \in (\varphi(d, \partial)\backslash\iota)$$

$$d :: \delta, (l \cdot \varphi\backslash_\iota).\mathrm{get}(), \sigma \hookrightarrow d :: \delta, \mathbf{weak}_\iota(\sigma(l)), \sigma \quad (get)$$
$$\text{if } \mathrm{get} \in (\varphi(d, \partial)\backslash\iota)$$

$$\delta, \mathsf{let}\, x = v \,\mathsf{in}\, e, \sigma \hookrightarrow \delta, e[v/x], \sigma \quad (let)$$

$$d :: \delta, \cdot v \cdot, \sigma \hookrightarrow \delta, v, \sigma \quad (pop)$$

$$\delta, E[e], \sigma \rightarrow \delta', E[e'], \sigma' \quad (context)$$
$$\text{if } \delta, e, \sigma \hookrightarrow \delta', e', \sigma'$$

Fig. 2. Operational semantics for pop

5. $(l \cdot \varphi \backslash_\iota).get()$ *where* $l \notin dom(\sigma)$
6. $(l \cdot \varphi \backslash_\iota).set(v)$ *where* $l \notin dom(\sigma)$

The reflexive, transitive closure of $\rightarrow$ is denoted $\rightarrow^\star$. Other language relevant to properties of evaluation is defined as follows.

Definition 5. *The domain* d_1 *is the* top-level domain. *An expression* e *is* top-level *if it contains no subexpressions of the form* $\cdot e' \cdot$ *or* $[\varrho].m(e')$ *or* $l \cdot \varphi \backslash_\iota$. *If* $d_1 :: nil, e, \varnothing \rightarrow^\star d_1 :: nil, v, \sigma$ *for top-level* e, *we say that* e evaluates to v. *If there does not exist* v *such that* e *evaluates to* v, *then* e diverges, *and if* $d_1 :: nil, e, \varnothing \rightarrow^\star d_1 :: nil, e', \sigma$ *and* $d_1 :: nil, e', \sigma$ *is stuck, then* e goes wrong.

An important feature of the pop semantics is that it grants every domain *at least* the default access rights to any object. Also, in the *send* rule, we always require a test to ensure that the active protection domain is authorized for the specified use of the object: this detail is the essence of our *use*-based, as opposed to *communication*-based, security model, in the sense that authorization for object access is checked when the object is used, not when it is communicated via message send or assignment. The **weak** mechanism semantics ensures that any return value from a message send to a weakened object is similarly weakened, and that the message send itself is allowable with respect to the weakening. The *cast* rule requires that any cast *restrict* access rights to a capability, so that increasing rights beyond the initial policy specification is disallowed. As we will see in Sect. 5, the pop type system statically enforces all of these checks, so that the authorization checks associated with casting, weakening, and message sends may be safely removed from the run-time system.

3.2.1 The self variable and self objects

In order for objects to always have complete access to themselves, the semantics specifies a rule for the use of self objects that imposes no run-time authorization checks; indeed, self objects have no interface or weakenings. The restriction that the variable **s** cannot appear unselected – that is, if **s** occurs in a program it must always be in an expression of the form $\mathbf{s}.m(e)$ – ensures that **s** cannot escape its own scope. This implies that giving **s** "full strength" is safe since it cannot provide a "back door" to the object by being communicated outside. Rights amplification via self, discussed in Sect. 2.6, is still possible, but this is a *feature* of capability-based security, not a flaw of the model.

4 Simulating pop

In this section we show how pop may be simulated in another language by a semantics-preserving transformation. The target language of this transformation comes preequipped with a sound let-polymorphic type system, so the transformation will later serve as the foundation for a pop type analysis (Sect. 5). This approach has distinct technical benefits: since the transformation is computable and easy to prove correct, a sound *indirect* type system for pop can immediately be obtained as the composition of the transformation and type judgements in the target language, eliminating the overhead of a type soundess proof entirely. The transformation also eases development of a *direct* pop type system – that is, where pop expressions are treated directly, rather than through transformation. This is because safety in a direct system can be demonstrated by a simple proof of correspondance between the direct and indirect type systems, rather than through the usual (and complicated) route of subject reduction. This technique has been used to good effect in previous static treatments of languages supporting stack-inspection security [20], information flow security [19], and elsewhere [23].

4.1 The target language: pml_B

The target language of our transformation is pml_B [25, 27], a calculus of extensible records based on Rémy's Projective ML [21] and equipped with references, sets, and set operations. The language pml_B allows definition of records with default values $\{v\}$, where for every label a we have $\{v\}.a = v$. Records r may be modified with the syntax $r\{a = v\}$ such that $(r\{a = v\}).a = v$ and $(r\{a = v\}).a' = r.a'$ for all other a'.

The language also allows definition of finite sets B of atomic identifiers b chosen from a countably infinite set $\mathcal{L}_b$, and cosets $\bar{B}$, denoting $\mathcal{L}_b \backslash B$. The language additionally contains set operations $\ni$, $\vee$, $\wedge$, and $\ominus$, which are membership check, union, intersection and difference operations, respectively.

The formal syntax for pml_B is defined in Fig. 3. The semantics of pml_B is defined in Fig. 4. Note that the one-step reduction relation $\rightarrow$ is defined on configurations (e, σ), where σ is a store. As for pop, the relation $\rightarrow^\star$ on pml_B configurations is the reflexive, transitive closure of $\rightarrow$.

4.2 The pop-*to*-pml_B *transformation*

We begin by defining a transformation of pop user interfaces into pml_B records with default values, denoted $\hat{\varphi}$, in Fig. 5. In words, interface definitions are encoded as rows with fields indexed by domain names, including the default domain. Also, for brevity, in the transformation definition we also define various syntactic sugarings in Fig. 5. The pop-to-pml_B transformation is then defined in Fig. 6. The translation is effected by transforming pop objects into rows with obj fields containing method transformations, ifc fields containing interface transformations, and strong fields containing sets denoting methods on which the object is *not* weak.

$$
\begin{array}{lll}
x \in \mathcal{V},\ a \in \mathcal{L}_a,\ b \in \mathcal{L}_b,\ B \subseteq \mathcal{L}_b & & \textit{identifiers} \\
v \ ::= \mathsf{fix}\,\mathbf{s}.\lambda x.e \mid s \mid \{v\} \mid v\{a = v\} \mid \mathrm{ref} \mid := \mid (:= l) \mid ! & & \textit{values} \\
s \ ::= B \mid \bar{B} \mid \vee \mid \wedge \mid \ominus \mid \ni_b & & \textit{sets, set operations} \\
e \ ::= x \mid v \mid e\,e \mid \mathsf{let}\,x = v\,\mathsf{in}\,e \mid \{e\} \mid e\{a = e\} \mid e.a & & \textit{expressions} \\
E ::= [\,] \mid E\,e \mid v\,E \mid \{E\} \mid E\{a = e\} \mid v\{a = E\} \mid E.a & & \textit{evaluation contexts}
\end{array}
$$

Fig. 3. Grammar for pml_B

$$
\begin{array}{rll}
(\mathsf{fix}\,\mathbf{s}.\lambda x.e)v, \sigma \to e[v/x][\mathsf{fix}\,\mathbf{s}.\lambda x.e/\mathbf{s}], \sigma & & (\beta) \\
\mathsf{let}\,x = v\,\mathsf{in}\,e, \sigma \to e[v/x], \sigma & & (\textit{let}) \\
\mathrm{ref}\ v, \sigma \to l, \sigma[l \mapsto v] & l \notin \mathrm{dom}(\sigma) & (\textit{ref}) \\
l := v, \sigma \to v, \sigma[l \mapsto v] & l \in \mathrm{dom}(\sigma) & (\textit{assign}) \\
!l, \sigma \to \sigma(l), \sigma & & (\textit{bang}) \\
\{v\}.a, \sigma \to v, \sigma & & (\textit{default}) \\
v_1\{a = v_2\}.a, \sigma \to v_2, \sigma & & (\textit{access}) \\
v_1\{a' = v_2\}.a, \sigma \to v_1.a, \sigma & a' \neq a & (\textit{skip}) \\
B \ni b, \sigma \to B, \sigma & \text{if } b \in B & (\textit{memcheck}) \\
B_1 \wedge B_2, \sigma \to B_1 \cap B_2, \sigma & & (\textit{intersect}) \\
B_1 \vee B_2, \sigma \to B_1 \cup B_2, \sigma & & (\textit{union}) \\
B_1 \ominus B_2, \sigma \to B_1 \backslash B_2, \sigma & & (\textit{difference}) \\
E[e], \sigma \to E[e'], \sigma' & \text{if } e, \sigma \to e', \sigma' & (\textit{context})
\end{array}
$$

Fig. 4. Operational semantics for pml_B

$$
\{d_1 \mapsto \iota_1, \ldots, \widehat{d_n \mapsto \iota_n}, \partial \mapsto \iota\} = \{\varnothing\}\{\partial = \iota\}\{d_1 = \iota_1\} \cdots \{d_n = \iota_n\}
$$

$$
\begin{array}{rll}
\{m_1 = e_1, \ldots, m_n = e_n\} \triangleq \{\varnothing\}\{m_1 = e_1\} \cdots \{m_n = e_n\} & & \\
\mathsf{fix}\,\mathbf{s}.\lambda\_.e \triangleq \mathsf{fix}\,\mathbf{s}.\lambda x.e & & x \text{ not free in } e \\
e_1; e_2 \triangleq \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 & & x \text{ not free in } e_2 \\
e \supseteq \iota \triangleq e \ni m_1; \ldots; e \ni m_n & & \iota = \{m_1, \ldots, m_n\}
\end{array}
$$

Fig. 5. pop-to-pml_B transformation auxiliary definitions

Of technical interest is the use of pml_B lambda abstractions with recursive binding to encode the self variable **s** in the transformation. Also of technical note is the manner in which weakenings are encoded. In a pop weakened object $\mathbf{weak}_\iota(o)$, the set ι denotes the methods that are inaccessible via weakening. In the encoding these sets are turned "inside out", so that the strong field in objects denotes the fields that *are* accessible; in an unweakened object definition, this field contains $\bar{\varnothing}$. Accordingly, in the translation of message sends, any resulting composition of weakenings is encoded as an *intersection* of the composed strong fields, rather than a union. We define the translation in this manner to allow a simple definition of set subtyping, as well as typings of set operations in the pml_B type system, which translate into a simpler direct type system for pop. See Sect. 5 for details.

The principal result of this section is the proof of correctness of the pop-to-pml_B transformation, in the sense that the transformation preserves program semantics. It is a simulation result which, aside from providing confidence in the faithfulness of the transformation, will allow us to immediately obtain an indirect type soundness result for pop based on soundness of the pml_B type system and will make direct type soundness for pop easier to prove as well. The desired property is stated as follows, and proved below:

Theorem 1 (pop-to-pml_B transformation correctness). *If e evaluates to v, then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_{d_1}$. If e diverges, then so does $[\![e]\!]_{d_1}$. If e goes wrong, then $[\![e]\!]_{d_1}$ goes wrong.*

4.3 Properties

Our proof of Theorem 1 will follow by induction on arbitrary computations in pop. However, to state the induction properly, it is necessary to extend the pop-to-pml_B transformation to run-time entities, as in Fig. 7. Note in

$$
\begin{aligned}
[\![x]\!]_d &= x \\
[\![\mathbf{s}.m(e)]\!]_d &= (\mathbf{s}\{\}.m)[\![e]\!]_d \\
[\![[m_i(x) = e_i\ ^{0<i\leq n}] \cdot d' \cdot \varphi\backslash_\iota]\!]_d &= \{\mathsf{obj} = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'}\ ^{0<i\leq n}\}, \\
&\quad\ \mathsf{ifc} = \hat{\varphi}, \\
&\quad\ \mathsf{strong} = \bar{\iota}\} \\
[\![e_1.m(e_2)]\!]_d &= \mathsf{let}\, os = \{o_1 = [\![e_1]\!]_d, o_2 = [\![e_2]\!]_d\}\ \mathsf{in} \\
&\quad\ \mathsf{let}\, i = os.o_1.\mathsf{ifc}\ \mathsf{in} \\
&\quad\ \mathsf{let}\, w = os.o_1.\mathsf{strong}\ \mathsf{in} \\
&\quad\ \mathsf{let}\, o = (os.o_1.\mathsf{obj})\{\}\ \mathsf{in} \\
&\quad\ ((i.d \vee i.\partial) \wedge w) \ni m; \\
&\quad\ \mathsf{let}\, o_3 = o.m(os.o_2)\ \mathsf{in} \\
&\quad\ o_3\{\mathsf{strong} = (w \wedge o_3.\mathsf{strong})\} \\
[\![e_{\backslash}(d', \iota)]\!]_d &= \mathsf{let}\, o_1 = [\![e]\!]_d\ \mathsf{in} \\
&\quad\ (o_1.\mathsf{ifc}.d' \vee o_1.\mathsf{ifc}.\partial) \supseteq \iota; \\
&\quad\ o_1\{\mathsf{ifc} = ((o_1.\mathsf{ifc})\{d' = \iota\})\} \\
[\![\mathbf{weak}_\iota(e)]\!]_d &= \mathsf{let}\, o_1 = [\![e]\!]_d\ \mathsf{in} \\
&\quad\ o_1\{\mathsf{strong} = (\bar{\iota} \wedge o_1.\mathsf{strong})\} \\
[\![\mathsf{ref}_\varphi e]\!]_d &= \mathsf{let}\, x = \mathsf{ref}\ [\![e]\!]_d\ \mathsf{in} \\
&\quad\ \mathsf{let}\, o = \lambda\_.\{\mathsf{get} = \lambda y.!x, \mathsf{set} = \lambda y.x := y\}\ \mathsf{in} \\
&\quad\ \{\mathsf{obj} = o, \mathsf{ifc} = \hat{\varphi}, \mathsf{strong} = \bar{\varnothing}\} \\
[\![\mathsf{let}\, x = v\ \mathsf{in}\, e]\!]_d &= \mathsf{let}\, x = [\![v]\!]_d\ \mathsf{in}\ [\![e]\!]_d
\end{aligned}
$$

Fig. 6. The pop-to-pml$_B$ term transformation

this definition, the transformation of values in stores may be parameterized by arbitrary domain labels d; the following lemma demonstrates that this is reasonable since the transformation of values $[\![v]\!]_d$ does not depend on d:

Lemma 2. *For all d and d', if $[\![v]\!]_d$ is defined, then $[\![v]\!]_d = [\![v]\!]_{d'}$.*

Proof. Immediate by definition of the transformation since for any case of v the identifier d does not appear in the RHS of the definition of $[\![v]\!]_d$. □

Next, we define two *substitution* lemmas relevant to the transformation:

Lemma 3. *If $[\![e]\!]_d$ is defined then $[\![e]\!]_d[[\![v]\!]_{d'}/x] = [\![e[v/x]]\!]_d$.*

Proof. By structural induction on e. In the basis we have $e = x'$, where by definition $[\![x]\!]_d = x'$. If $x' \neq x$, then:

$$[\![e]\!]_d[[\![v]\!]_{d'}/x] = [\![e[v/x]]\!]_d = x' .$$

Otherwise we have $[\![x]\!]_d[[\![v]\!]_{d'}/x] = [\![v]\!]_{d'}$; but $[\![v]\!]_{d'} = [\![v]\!]_d$ by Lemma 2, and $[\![x[v/x]]\!]_d = [\![v]\!]_d$, so this case holds. The other cases follow in a straightforward manner by the induction hypothesis. □

Lemma 4. *Let:*

$$s = [m_i(x) = e_i\ ^{0<i\leq n}]$$
$$v = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d\ ^{0<i\leq n}\} .$$

Then if $[\![e]\!]_d$ is defined, $[\![e]\!]_d[v/\mathbf{s}] = [\![e[s/\mathbf{s}]]\!]_d$.

Proof. By structural induction on e. In the basis we have $e = x$ such that $x \neq \mathbf{s}$, since $[\![\mathbf{s}]\!]_d$ is undefined, so that $x[s/\mathbf{s}] = x$ and $[\![x]\!]_d = x$ by definition, therefore:

$$[\![e]\!]_d[v/\mathbf{s}] = [\![e[s/\mathbf{s}]]\!]_d = x .$$

The induction step proceeds by case analysis on e. In case $e = \mathbf{s}.m(e')$, we have that:

$$e[s/\mathbf{s}] = [m_i(x) = e_i\ ^{0<i\leq n}].m(e'[s/\mathbf{s}])$$

by definition of self substitution, and:

$$[\![[m_i(x) = e_i\ ^{0<i\leq n}].m(e')]\!]_d$$
$$=$$
$$((\mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d\ ^{0<i\leq n}\})\{\}.m)[\![e'[s/\mathbf{s}]]\!]_d$$

by definition of the transformation. But:

$$[\![\mathbf{s}.m(e')]\!]_d = (\mathbf{s}\{\}.m)[\![e']\!]_d$$

so that:

$$((\mathbf{s}\{\}.m)[\![e']\!]_d)[v/\mathbf{s}]$$
$$=$$
$$((\mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d\ ^{0<i\leq n}\})\{\}.m)([\![e']\!]_d[v/\mathbf{s}])$$

and $[\![e'[s/\mathbf{s}]]\!]_d = [\![e']\!]_d[v/\mathbf{s}]$ by the induction hypothesis, so this case holds. The other cases follow in a straightforward manner by the induction hypothesis. □

We may now prove the core of our simulation result by showing that one-step reductions $\hookrightarrow$ may be simulated via the transformation.

$$
\begin{aligned}
[\![[\,]]\!]_\delta &= [\,] \\
[\![[m_i(x) = e_i\ ^{0<i\leq n}].m(E)]\!]_{d::\delta} &= ((\mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d\ ^{0<i\leq n}\})\{\}.m)[\![E]\!]_{d::\delta} \\
[\![E.m(e)]\!]_{d::\delta} &= \mathsf{let}\, os = \{o_1 = [\![E]\!]_{d::\delta}, o_2 = [\![e]\!]_d\}\ \mathsf{in} \\
&\quad \mathsf{let}\, i = os.o_1.\mathrm{ifc}\ \mathsf{in} \\
&\quad \mathsf{let}\, w = os.o_1.\mathrm{strong}\ \mathsf{in} \\
&\quad \mathsf{let}\, o = (os.o_1.\mathrm{obj})\{\}\ \mathsf{in} \\
&\quad ((i.d \vee i.\partial) \wedge w) \ni m; \\
&\quad \mathsf{let}\, o_3 = o.m(os.o_2)\ \mathsf{in} \\
&\quad o_3\{\mathrm{strong} = (w \wedge o_3.\mathrm{strong})\} \\
[\![v.m(E)]\!]_{d::\delta} &= \mathsf{let}\, os = \{o_1 = [\![v]\!]_d, o_2 = [\![E]\!]_{d::\delta}\}\ \mathsf{in} \\
&\quad \mathsf{let}\, i = os.o_1.\mathrm{ifc}\ \mathsf{in} \\
&\quad \mathsf{let}\, w = os.o_1.\mathrm{strong}\ \mathsf{in} \\
&\quad \mathsf{let}\, o = (os.o_1.\mathrm{obj})\{\}\ \mathsf{in} \\
&\quad ((i.d \vee i.\partial) \wedge w) \ni m; \\
&\quad \mathsf{let}\, o_3 = o.m(os.o_2)\ \mathsf{in} \\
&\quad o_3\{\mathrm{strong} = (w \wedge o_3.\mathrm{strong})\} \\
[\![e\,|\,(d,\iota)]\!]_\delta &= \mathsf{let}\, o_1 = [\![E]\!]_\delta\ \mathsf{in} \\
&\quad (o_1.\mathrm{ifc}.d \vee o_1.\mathrm{ifc}.\partial) \supseteq \iota; \\
&\quad o_1\{\mathrm{ifc} = ((o_1.\mathrm{ifc})\{d = \iota\})\} \\
[\![\mathbf{weak}_\iota(E)]\!]_\delta &= \mathsf{let}\, o_1 = [\![E]\!]_\delta\ \mathsf{in} \\
&\quad o_1\{\mathrm{strong} = (\bar{\iota} \wedge o_1.\mathrm{strong})\} \\
[\![\mathrm{ref}_\varphi E]\!]_\delta &= \mathsf{let}\, x = \mathrm{ref}\ [\![E]\!]_\delta\ \mathsf{in} \\
&\quad \mathsf{let}\, o = \lambda\_.\{\mathrm{get} = \lambda y.!x, \mathrm{set} = \lambda y.x := y\}\ \mathsf{in} \\
&\quad \{\mathrm{obj} = o, \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\varnothing}\} \\
[\![\cdot E \cdot]\!]_{d::\delta} &= [\![E]\!]_\delta \\
\\
[\![[m_i(x) = e_i\ ^{0<i\leq n}].m(e)]\!]_d &= ((\mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_d\ ^{0<i\leq n}\})\{\}.m)[\![e]\!]_d \\
[\![l \cdot \varphi\backslash_\iota]\!]_d &= \{\,\mathrm{obj} = \lambda\_.\{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\}, \\
&\qquad \mathrm{ifc} = \hat{\varphi}, \\
&\qquad \mathrm{strong} = \bar{\iota}\} \\
[\![\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}]\!] &= \{l_1 \mapsto [\![v_1]\!]_d, \ldots, l_n \mapsto [\![v_n]\!]_d\}
\end{aligned}
$$

Fig. 7. The pop-to-pml$_B$ evaluation context and run-time term transformations

Lemma 5. *The following assertions hold:*

1. *If* $d :: \delta, e_1, \sigma \hookrightarrow d' :: d :: \delta, \cdot e_2 \cdot, \sigma$ *by send, then* $[\![e_1]\!]_d, [\![\sigma]\!] \rightarrow^\star [\![e_2]\!]_{d'}, [\![\sigma]\!]$.
2. *If* $d' :: d :: \delta, \cdot v \cdot, \sigma \hookrightarrow d :: \delta, v, \sigma$ *by pop, then* $[\![v]\!]_d, [\![\sigma]\!] \rightarrow^\star [\![v]\!]_{d'}, [\![\sigma]\!]$.
3. *If* $d :: \delta, e_1, \sigma_1 \hookrightarrow d :: \delta, e_2, \sigma_2$ *by some rule besides send or pop, then* $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e_2]\!]_d, [\![\sigma_2]\!]$.

Proof. Each assertion is treated individually:

1. In this case by definition of *send* we have $e_1 = ([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v)$, where $\varrho = (m_i(x) = e_i\ ^{0<i\leq n})$ and $(m(x) = e) \in \varrho$, and we also have $e_2 = \mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x])$ and $m \in (\varphi(d,\partial)\backslash\iota)$; therefore $m \notin \iota$ and $m \in \varphi(d,\partial)$. But by definition of the transformation we have:

$$
\begin{aligned}
[\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d = \{&\mathrm{obj} = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'}\ ^{0<i\leq n}\}, \\
&\mathrm{ifc} = \hat{\varphi}, \\
&\mathrm{strong} = \bar{\iota}\}
\end{aligned}
$$

therefore, letting $v' = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e_i]\!]_{d'}\ ^{0<i\leq n}\}$ we have:

$$[\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\mathrm{ifc}, [\![\sigma]\!] \rightarrow \hat{\varphi}, [\![\sigma]\!]$$

$$[\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\mathrm{strong}, [\![\sigma]\!] \rightarrow \bar{\iota}, [\![\sigma]\!]$$

$$
([\![[\varrho] \cdot d' \cdot \varphi\backslash_\iota]\!]_d.\mathrm{obj})\{\}, [\![\sigma]\!] \rightarrow^\star \{m_i = \lambda x.[\![e_i]\!]_{d'}[v'/\mathbf{s}]\ ^{0<i\leq n}\}, [\![\sigma]\!]\,.
$$

Further, since $m \notin \iota$ and $m \in \varphi(d,\partial)$, it is the case that $m \in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and $m \in \bar{\iota}$, i.e., $m \in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that $[\![e]\!]_d, [\![\sigma]\!] \rightarrow^\star e'', [\![\sigma]\!]$ in this case, where:

$$
\begin{aligned}
e'' = &\ \mathsf{let}\, o_3 = [\![e]\!]_{d'}[v'/\mathbf{s}][[\![v]\!]_d/x]\ \mathsf{in} \\
&(o_3\{\mathrm{strong} = (\bar{\iota} \wedge o_3.\mathrm{strong})\})
\end{aligned}
$$

by definition of the transformation and pml$_B$ reduction. But by Lemmas 3 and 4 we have that:

$$([\![e]\!]_{d'}[v'/\mathbf{s}][[\![v]\!]_d/x]) = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_{d'}$$

so:

$$
\begin{aligned}
e'' = &\ \mathsf{let}\, o_3 = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_{d'}\ \mathsf{in} \\
&(o_3\{\mathrm{strong} = (\bar{\iota} \wedge o_3.\mathrm{strong})\})
\end{aligned}
$$

and e'' is equivalent to $[\![\mathbf{weak}_\iota(e[[\varrho]/\mathbf{s}][v/x])]\!]_{d'}$, that is, to $[\![e_2]\!]_{d'}$, so the assertion holds.

2. This assertion holds immediately by Lemma 2 and reflexivity of $\rightarrow$.

3. This assertion follows by case analysis on the remaining reduction rules.

Case *self*. In this case $\sigma_1 = \sigma_2$ and $e_1 = [\varrho].m(v)$, where $\varrho = (m_i(x) = e_i \ ^{0<i\leq n})$ and $(m(x) = e) \in \varrho$, and $e_2 = e[[\varrho]/\mathbf{s}][v/x]$. Let:

$$v' = \mathsf{fix}\,\mathbf{s}.\lambda\_.\{m_i = \lambda x.[\![e]\!]_d \ ^{0<i\leq n}\}\,.$$

Then by definition of the transformation we have $[\![e_1]\!]_d = (v'\{\}.m)[\![v]\!]_d$, therefore:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e]\!]_d[v'/\mathbf{s}][[\![v]\!]_d/x], [\![\sigma_2]\!]\,.$$

But by Lemmas 3 and 4 we have:

$$[\![e]\!]_d[v'/\mathbf{s}][[\![v]\!]_d/x] = [\![e[[\varrho]/\mathbf{s}][v/x]]\!]_d = [\![e_2]\!]_d$$

so this case holds.

Case *cast*. In this case $\sigma_1 = \sigma_2$ and $e_1 = (co \cdot \varphi\backslash_\iota)_! (d', \iota')$ and $e_2 = (co \cdot (\varphi[d' \mapsto \iota'])\backslash_\iota)$, with $\iota' \subseteq \varphi(d', \partial)$. Let $e = (co \cdot \varphi\backslash_\iota)$; then there exists e' such that:

$$[\![e]\!]_d = \{\mathrm{obj} = e', \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\iota}\}\,,$$
$$[\![e_2]\!]_d = \{\mathrm{obj} = e', \mathrm{ifc} = \varphi[\widehat{d' \mapsto \iota'}], \mathrm{strong} = \bar{\iota}\}$$

by definition of the transformation. Further:

$$[\![e_1]\!]_d = \mathsf{let}\, o_1 = [\![e]\!]_d\ \mathsf{in}\\ (o_1.\mathrm{ifc}.d' \vee o_1.\mathrm{ifc}.\partial) \supseteq \iota';\\ o_1\{\mathrm{ifc} = ((o_1.\mathrm{ifc})\{d' = \iota'\})\}\,.$$

Let $o_1 = [\![e]\!]_d$. Since $\iota' \subseteq \varphi(d', \partial)$, therefore $\iota \subseteq (o_1.\mathrm{ifc}.d' \cup o_1.\mathrm{ifc}.\partial)$, and since $o_1.\mathrm{ifc}, [\![\sigma_2]\!] \rightarrow^\star \hat{\varphi}, [\![\sigma_2]\!]$, we have:

$$(o_1.\mathrm{ifc})\{d' = \iota'\}, [\![\sigma_2]\!] \rightarrow^\star (\varphi[\widehat{d' \mapsto \iota'}]), [\![\sigma_2]\!]$$

etc.; therefore $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e_2]\!]_d, [\![\sigma_2]\!]$ in this case by definition of $\rightarrow^\star$.

Case *weaken*. In this case $\sigma_1 = \sigma_2$, $e_1 = \mathbf{weak}_\iota(co \cdot \varphi\backslash_{\iota'})$ and $e_2 = co \cdot \varphi\backslash_{(\iota \cup \iota')}$. Let $e = co \cdot \varphi\backslash_{\iota'}$; then by definition of the transformation there exists e' such that:

$$[\![e]\!]_d = \{\mathrm{obj} = e', \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\iota}\}$$
$$[\![e_2]\!]_d = \{\mathrm{obj} = e', \mathrm{ifc} = \widehat{\varphi}, \mathrm{strong} = \bar{\iota'} \cap \bar{\iota}\}$$

and:

$$[\![e_1]\!]_d = \mathsf{let}\, o_1 = [\![e]\!]_d\ \mathsf{in}\ o_1\{\mathrm{strong} = (\bar{\iota'} \wedge o_1.\mathrm{strong})\}$$

so clearly $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow [\![e_2]\!]_d, [\![\sigma_2]\!]$ in this case.

Case *newcell*. In this case $e_1 = \mathrm{ref}_\varphi v$, $e_2 = l \cdot \varphi\backslash_\varnothing$, and $\sigma_2 = \sigma_1[l \mapsto v]$, where $l \notin \mathrm{dom}(\sigma_1)$. By definition of the transformation we have:

$$[\![e_1]\!]_d = \mathsf{let}\, x = \mathrm{ref}\, [\![v]\!]_d\ \mathsf{in}\\ \mathsf{let}\, o = \lambda\_.\{\mathrm{get} = \lambda y.!x, \mathrm{set} = \lambda y.x := y\}\ \mathsf{in}\\ \{\mathrm{obj} = o, \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\varnothing}\}$$

$$[\![e_2]\!]_d = \{\mathrm{obj} = \lambda\_.\{\mathrm{get} = \lambda y.!l,\\ \mathrm{set} = \lambda y.l := y\},\\ \mathrm{ifc} = \hat{\varphi}, \mathrm{strong} = \bar{\varnothing}\}$$

Now, since $l \notin \mathrm{dom}(\sigma)$, therefore $l \notin \mathrm{dom}([\![\sigma]\!])$, so by definition of $\mathrm{pml}_B \rightarrow^\star$:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e_2]\!]_d, [\![\sigma_1]\!][l \mapsto [\![v]\!]_d]$$

But $[\![\sigma_1]\!][l \mapsto [\![v]\!]_d] = [\![\sigma_1[l \mapsto v]]\!]$ by definition and Lemma 2, so this case holds.

Case *set*. In this case $e_1 = (l \cdot \varphi\backslash_\iota).\mathrm{set}(v)$, where $l \in \mathrm{dom}(\sigma)$ and $\mathrm{set} \in (\varphi(d, \partial)\backslash\iota)$, $e_2 = \mathbf{weak}_\iota(v)$ and $\sigma_2 = \sigma_1[l \mapsto v]$. Then by definition of the transformation we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d = \{\,\mathrm{obj} = \lambda\_.\{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\},\\ \mathrm{ifc} = \hat{\varphi},\\ \mathrm{strong} = \bar{\iota}\}$$

so by definition of $\rightarrow^\star$ we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{ifc}, [\![\sigma]\!] \rightarrow\ \hat{\varphi}, [\![\sigma]\!]\\ [\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{strong}, [\![\sigma]\!] \rightarrow\ \bar{\iota}, [\![\sigma]\!]\\ ([\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{obj})\{\}, [\![\sigma]\!] \rightarrow^\star\\ \{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\}, [\![\sigma]\!]$$

Further, since $\mathrm{set} \in (\varphi(d, \partial)\backslash\iota)$, therefore $\mathrm{set} \notin \iota$ and $\mathrm{set} \in \varphi(d, \partial)$, so it is the case that $\mathrm{set} \in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and $\mathrm{set} \in \bar{\iota}$, i.e., $\mathrm{set} \in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that:

$$[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star e'', [\![\sigma_1]\!][l \mapsto [\![v]\!]_d]$$

in this case, where:

$$e'' = \mathsf{let}\, o_3 = [\![v]\!]_d\ \mathsf{in}\ (o_3\{\mathrm{strong} = (\bar{\iota} \wedge o_3.\mathrm{strong})\})$$

by definition of the transformation and pml_B reduction. But e'' is equivalent to $[\![e_2]\!]_d$, and $[\![\sigma_1]\!][l \mapsto [\![v]\!]_d] = [\![\sigma_1[l \mapsto v]]\!]_d$ by definition, so this case holds.

Case *get*. In this case $e_1 = (l \cdot \varphi\backslash_\iota).\mathrm{get}()$, where $\mathrm{get} \in (\varphi(d, \partial)\backslash\iota)$, $\sigma_1 = \sigma_2$ and $e_2 = \mathbf{weak}_\iota(\sigma_2(l))$. Then by definition of the transformation we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d = \{\,\mathrm{obj} = \lambda\_.\{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\},\\ \mathrm{ifc} = \hat{\varphi},\\ \mathrm{strong} = \bar{\iota}\}$$

so by definition of $\rightarrow^\star$ we have:

$$[\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{ifc}, [\![\sigma]\!] \rightarrow\ \hat{\varphi}, [\![\sigma]\!]\\ [\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{strong}, [\![\sigma]\!] \rightarrow\ \bar{\iota}, [\![\sigma]\!]\\ ([\![l \cdot \varphi\backslash_\iota]\!]_d.\mathrm{obj})\{\}, [\![\sigma]\!] \rightarrow^\star\\ \{\mathrm{get} = \lambda y.!l, \mathrm{set} = \lambda y.l := y\}, [\![\sigma]\!]$$

Further, since $\mathrm{get} \in (\varphi(d, \partial)\backslash\iota)$, therefore $\mathrm{get} \notin \iota$ and $\mathrm{get} \in \varphi(d, \partial)$, so it is the case that $\mathrm{get} \in (\hat{\varphi}.d \cup \hat{\varphi}.\partial)$ and $\mathrm{get} \in \bar{\iota}$, i.e., $\mathrm{get} \in ((\hat{\varphi}.d \cup \hat{\varphi}.\partial) \cap \bar{\iota})$. Therefore we have that $[\![e_1]\!]_d, [\![\sigma_1]\!] \rightarrow^\star e'', [\![\sigma_2]\!]$ in this case, where:

$$e'' = \mathsf{let}\, o_3 = [\![\sigma_2]\!](l)\ \mathsf{in}\ (o_3\{\mathrm{strong} = (\bar{\iota} \wedge o_3.\mathrm{strong})\})$$

by definition of the transformation and pml_B reduction. Let $v = \sigma_2(l)$; then $[\![\sigma_2]\!](l) = [\![v]\!]_d$ by definition and

Lemma 2, so e'' is equivalent to $[\![e_2]\!]_d$ by definition; therefore this case holds.

Case *let* follows trivially by Lemma 3. □

Before turning to arbitrary-length computations, we state another result relevant to the transformation of values. The proof is immediate by definition of the transformation.

Lemma 6. *For all v, if v is a closed value and $[\![v]\!]_d$ is defined, then $[\![v]\!]_d$ is a value.*

Note that the only value form on which the transformation is undefined is $[\varrho]$; since unselected self is disallowed in programs, it is not necessary to treat this case.

Now we consider arbitrary-length computations with respect to $\rightarrow^\star$. To perform the necessary analysis, we extend the pop-to-pml$_B$ transformation to evaluation contexts in a straightforward manner. The context transformation is defined in Fig. 7; anticipating our technique, we will apply transformations to contexts along with the *reverse* of domain stacks in a configuration, since the oldest stack frames will apply to the outermost variables in contexts. We note that the current transformation is indeed a transformation from contexts to contexts:

Lemma 7. *For all closed E, any defined $[\![E]\!]_\delta$ is a well-formed evaluation context.*

Proof. Immediate by definition of the context transformation; the only mildly interesting case is $E = v.m(E)$, but in this case $[\![E]\!]_\delta$ is well-formed by Lemma 6. □

We then state some relevant properties of the context transformation. The proof of these properties follows by a straightforward induction on evalution contexts; it is omitted here since it is lengthy and uninteresting, but it is given in [25]:

Lemma 8. *The following properties hold:*

1. *If $[\![E[e]]\!]_d$ is defined, then the frame depth of E is* 0.
2. *If $[\![E[e]]\!]_d$ is defined, then $[\![E[e]]\!]_d = [\![E]\!]_{d::nil}[\![e]\!]_d]$.*
3. *If $E = E_1[E_2]$ where the frame depth of E_1 equals the length of δ, and $[\![E]\!]_{\mathrm{rev}(d::\delta)}$ is defined, then $[\![E]\!]_{\mathrm{rev}(d::\delta)} = [\![E_1]\!]_{\mathrm{rev}(d::\delta)}[\![E_2]\!]_{d::nil}]$.*

Next, we define a simulation relation between pop and pml$_B$ based on the expression and context transformations:

Definition 6. *For all $d :: \delta$,* pop *expressions e, and* pml$_B$ *expressions e', the relation $(d :: \delta, e) \lhd e'$ holds iff there exists E_1 and e_1 such that $e = E_1[e_1]$, the frame depth of E_1 equals the length of δ, and $e' = [\![E_1]\!]_{\mathrm{rev}(d::\delta)}[\![e_1]\!]_d]$.*

We then prove that this relation is a mapping:

Lemma 9. *If $(\delta, e) \lhd e'$ and $(\delta, e) \lhd e''$, then $e' = e''$.*

Proof. Let $\delta = (d :: \delta')$, and let E_1, E_1', e_1, and e_1' be such that $E_1[e_1] = E_1'[e_1'] = e$, with the frame depths of E_1 and E_1' equal to the length of δ' and $[\![E_1]\!]_{\mathrm{rev}(d::\delta')}[\![e_1]\!]_d] = e'$ and $[\![E_1']\!]_{\mathrm{rev}(d::\delta')}[\![e_1']\!]_d] = e''$. Assume w.l.o.g. that $e_1 = E[e_1']$ for some E, so that $E_1' = E_1[E]$. Since $[\![e_1]\!]_d$ is defined, therefore the frame depth of E is 0 by Lemma 8, so also by Lemma 8 we have that $[\![e_1]\!]_d = [\![E]\!]_{d::nil}[\![e_1']\!]_d]$. Further, since the frame depth of E_1 equals the length of δ', therefore $[\![E_1']\!]_{\mathrm{rev}(d::\delta')} = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[\![E]\!]_{d::nil}]$ by Lemma 8. But then $[\![E_1']\!]_{\mathrm{rev}(d::\delta')}[\![e_1']\!]_d] = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[\![E]\!]_{d::nil}[\![e_1]\!]_d]] = [\![E_1]\!]_{\mathrm{rev}(d::\delta')}[\![e_1]\!]_d]$, therefore $e' = e''$. □

The following result shows that the simulation relation is preserved by one step of pop reduction:

Lemma 10. *If $\delta_1, e_1, \sigma_1 \rightarrow \delta_2, e_2, \sigma_2$ and $(\delta_1, e_1) \lhd e_1'$, then $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ such that $(\delta_2, e_2) \lhd e_2'$.*

Proof. By *context* we have that $e_1 = E[e]$ and $e_2 = E[e']$ with $\delta_1, e, \sigma_1 \hookrightarrow \delta_2, e', \sigma_2$. The proof then proceeds by cases corresponding to those treated in the assertions enumerated in Lemma 5:

Case 1. In this case $\delta_1 = (d :: \delta)$, $\delta_2 = (d' :: d :: \delta)$, $\sigma_1 = \sigma_2$ and e' is of the form $\cdot e'' \cdot$ with $[\![e]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e'']\!]_{d'}, [\![\sigma_2]\!]$. Let $e_1' = [\![E]\!]_{\mathrm{rev}(\delta_1)}[\![e]\!]_d]$ and $e_2' = [\![E[\cdot\,[\,]\cdot]]\!]_{\mathrm{rev}(\delta_2)}[\![e'']\!]_{d'}]$. The frame depth of E equals the length of δ by Corollary 1, so also the frame depth of $E[\cdot\,[\,]\cdot]$ equals the length of $d :: \delta$; therefore we have that $(\delta_1, e_1) \lhd e_1'$ and $(\delta_2, e_2) \lhd e_2'$ by definition. But clearly $[\![E[\cdot\,[\,]\cdot]]\!]_{\mathrm{rev}(\delta_2)} = [\![E]\!]_{\mathrm{rev}(\delta_1)}$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by multiple applications of *context*.

Case 2. In this case $\delta_1 = (d' :: d :: \delta)$, $\delta_2 = (d :: \delta)$, $\sigma_1 = \sigma_2$, and e is of the form $\cdot v \cdot$ and $e' = v$. Let $e_1' = [\![E[\cdot\,[\,]\cdot]]\!]_{\mathrm{rev}(\delta_1)}[\![v]\!]_{d'}]$ and $e_2' = [\![E]\!]_{\mathrm{rev}(\delta_2)}[\![v]\!]_d]$. The frame depth of $E[\cdot\,[\,]\cdot]$ equals the length of δ_2 by well-formedness of configurations, so also the frame depth of E equals the length of δ; therefore we have that $(\delta_1, e_1) \lhd e_1'$, and $(\delta_2, e_2) \lhd e_2'$ by definition. But clearly $[\![E[\cdot\,[\,]\cdot]]\!]_{\mathrm{rev}(\delta_1)} = [\![E]\!]_{\mathrm{rev}(\delta_2)}$, and $[\![v]\!]_{d'} = [\![v]\!]_d$ by Lemma 2, hence $e_1' = e_2'$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by reflexivity of $\rightarrow^\star$.

Case 3. In this case $\delta_1 = \delta_2 = d :: \delta$, with $[\![e]\!]_d, [\![\sigma_1]\!] \rightarrow^\star [\![e']\!]_d, [\![\sigma_2]\!]$ by Lemma 5. Let $e_1' = [\![E]\!]_{\mathrm{rev}(\delta_1)}[\![e]\!]_d]$ and $e_2' = [\![E]\!]_{\mathrm{rev}(\delta_2)}[\![e']\!]_d]$. By definition of $\hookrightarrow$ both e and e' are unframed, so the frame depth of E in this case is equal to the length of δ by Corollary 1, hence $(\delta_1, e_1) \lhd e_1'$, and $(\delta_2, e_2) \lhd e_2'$ by definition. Furthermore, since $\delta_1 = \delta_2$ we have that $[\![E]\!]_{\mathrm{rev}(\delta_1)} = [\![E]\!]_{\mathrm{rev}(\delta_2)}$, so $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$ in this case by multiple applications of *context*. □

The previous result then generalizes easily to arbitrary computations since the simulation relation is a mapping:

Lemma 11. *If $\delta_1, e_1, \sigma_1 \rightarrow^\star \delta_2, e_2, \sigma_2$ and $(\delta_1, e_1) \lhd e_1'$, then $e_1', [\![\sigma_1]\!] \rightarrow^\star e_2', [\![\sigma_2]\!]$, where $(\delta_2, e_2) \lhd e_2'$.*

Proof. Straightforward by Lemma 10 and induction on the length of the reduction $\delta_1, e_1, \sigma_1 \rightarrow^\star \delta_2, e_2, \sigma_2$. □

Before proving the main result we make one final observation, that relevant dynamic properties of configurations are preserved by transformation:

Lemma 12. *If δ, e, σ is stuck and $(\delta, e) \lhd e'$, then $e', [\![\sigma]\!]$ goes wrong. If $(\delta, v) \lhd e'$, then e' is a value. If $(\delta, e) \lhd e'$ and e' is not a value nor of the form $\cdot v \cdot$, then e' is not a value.*

Proof. Suppose δ, e, σ is stuck; then $e = E[e']$, where e' is one of the forms specified in Lemma 1. For each form, it is easy to see that the transformation $[\![e']\!]_d, [\![\sigma]\!]$ will go wrong, and here we only sketch the relevant case analysis: if e is stuck because e' is a method select on m that is unauthorized to the active domain or that has been disallowed by weakening, then the transformation implements a check that will also fail. If e is stuck because e' is a method select on m that does not exist in the object, then an m field will not exist in $[\![e']\!]_d$, so a projection of that field will fail. If e is stuck because e' is a set or a get on a cell object with location l such that $l \notin \sigma$, then $l \notin [\![\sigma]\!]$, so the transformation of these actions will also fail, as the transformation preserves store locations.

Suppose that $(\delta, v) \lhd e'$; then $e' = [\![v]\!]_d$, where $\delta = d :: \delta$ by definition of $\lhd$, and $[\![v]\!]_d$ is a value by Lemma 6.

Finally, suppose e is not a value nor of the form $\cdot v \cdot$. Let $\delta = d :: \delta$; since $(\delta, e) \lhd e'$, there exists E and e'' such that $e = E[e'']$ and $e' = [\![E]\!]_{\mathrm{rev}(\delta)}[\![e'']\!]_d$ by definition. Suppose that $E = [\,]$; then e'' is not a value, and clearly $[\![e'']\!]_d$ is not a value by definition of the transformation. Suppose E is composite; then clearly $[\![E]\!]_{\mathrm{rev}(\delta)}[\![e'']\!]_d$ is not a value by definition of the transformation, since E is not of the form $\cdot[\,]\cdot$ by assumption, and for any e''', $[\![E]\!]_{\mathrm{rev}(\delta)}[e''']$ is not a value in this case. □

We can now restate and prove the principal result of this section, that is, the correctness of the pop-to-pml$_B$ transformation, as follows:

Theorem 1 (pop-to-pml$_B$ transformation correctness). *If e evaluates to v, then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_d$. If e diverges, then so does $[\![e]\!]_{d_1}$. If e goes wrong, then $[\![e]\!]_{d_1}$ goes wrong.*

Proof. Suppose for top-level e we have $d_1 :: nil, e, \varnothing \rightarrow^\star d_1 :: nil, v, \sigma$. Then $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ and $(d_1 :: nil, v) \lhd [\![v]\!]_{d_1}$ by definition, and $[\![e]\!]_{d_1}, \varnothing \rightarrow^\star [\![v]\!]_{d_1}, [\![\sigma]\!]$ by Lemmas 11 and 9, and $[\![v]\!]_{d_1}$ is a value by Lemma 12.

Suppose for top-level e we have that $d_1 :: nil, e, \varnothing$ does not terminate, and suppose on the contrary that there exists σ and v such that $[\![e]\!]_{d_1}, \varnothing \rightarrow^\star v, \sigma$. Since $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ by definition, by Lemma 11 there must exist e', σ', and δ such that $(\delta, e') \lhd v$ and $\sigma = [\![\sigma']\!]$ and $d_1 :: nil, e, \varnothing \rightarrow^\star \delta, e', \sigma'$, where e' is not a value nor of the form $\cdot v' \cdot$ by assumption, since in the latter case δ, e', σ' would evaluate to v' by definition of $\rightarrow^\star$ and well-formedness of configurations. But then v is not a value by Lemma 12, which is a contradiction.

Finally, suppose for top-level e we have $d_1 :: nil, e$, $\varnothing \rightarrow^\star \delta, e', \sigma$ and δ, e', σ is stuck. Since $(d_1 :: nil, e) \lhd [\![e]\!]_{d_1}$ by definition, therefore $[\![e]\!]_{d_1}, \varnothing \rightarrow^\star e'', [\![\sigma]\!]$ such that $(\delta, e') \lhd e''$ by Lemma 11, and $e'', [\![\sigma]\!]$ goes wrong by Lemma 12. □

5 Types for pop

In this section we develop a static type analysis for pop on the foundation of the pop-to-pml$_B$ transformation and the pml$_B$ type system. We will define an *indirect* type system for pop based entirely on the pml$_B$ type system. We will also develop a *direct* type system for pop by defining a new set of type judgement rules expressly for pop terms and proving type safety in this system. As will become clear, even the direct type analysis will benefit from the pop-to-pml$_B$ transformation, thanks to a proof technique that renders an ab initio demonstration of subject reduction unneccesary to obtain type safety. Principal benefits of the pop type analysis are the optimizations that may be effected as a result of type safety, which we also discuss.

While we focus on the logical type system in this presentation, we will briefly describe how the transformational approach has benefits for type *inference*, allowing an algorithm to be developed using existing, efficient methods.

5.1 Indirect types

For a thorough treatment of the definition and soundness proof of the pml$_B$ type system, the reader is directed to [25, 27]. Here, we give a succinct overview. A sound polymorphic type system for pml$_B$ is obtained in a straightforward manner as an instantiation of HM(X) [17, 26], a constraint-based polymorphic type framework. Type judgements in HM(X) are of the form $C, \Gamma \vdash e : \sigma$, where C is a type constraint set, Γ is a typing environment, and σ is a polymorphic type scheme. The instantiation consists of a type language including row types [21] and a specialized language of *set* types, defined in Fig. 8. To ensure that only meaningful types can be built, we immediately equip this type language with *kinding* rules, defined in Fig. 9, and hereafter consider only well-kinded types. Note in particular that these kinding rules disallow duplication of record field and set element labels.

Set types behave in a manner similar to row types but have an abbreviated form more appropriate for application to sets. In fact, set types have a direct interpretation as a particular form of row types [27]. The field constructors $+$ and $-$ denote whether a set element is present or absent, respectively. The set types $\varnothing$ and ω behave similarly

$$\begin{array}{lll} \tau ::= \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \tau \text{ ref} \mid \{\tau\} \mid & & \\ \quad a : \tau\,;\ \tau \mid \partial\tau \mid b\tau, \tau \mid \varnothing \mid \omega \mid c & & \textit{types} \\ c ::= + \mid - & & \textit{constructors} \end{array}$$

Fig. 8. pml$_B$ type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau : \mathit{Type}}{\tau \text{ ref} : \mathit{Type}} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}(\tau)_\varnothing, \mathit{Row}(c)_\varnothing}{\{\tau\} : \mathit{Type}} \qquad \varnothing : \mathit{Row}(c)_B \qquad \omega : \mathit{Row}(c)_B \qquad c : \mathit{Con}$$

$$\frac{\tau : \mathit{Con} \qquad b \notin B \qquad \tau' : \mathit{Row}(c)_{B \cup \{b\}}}{(b\tau, \tau') : \mathit{Row}(c)_B} \qquad \frac{\tau : \mathit{Type}}{\partial\tau : \mathit{Row}(\tau)_A} \qquad \frac{\tau : \mathit{Type} \qquad a \notin A \qquad \tau' : \mathit{Row}(\tau)_{A \cup \{a\}}}{(a : \tau\,;\ \tau') : \mathit{Row}(\tau)_A}$$

Fig. 9. Kinding rules for pml_B types

to the uniform row constructor $\partial\tau$; the type $\varnothing$ (resp. ω) specifies that all other elements not explicitly mentioned in the set type are absent (resp. present). For example, the set $\{b_1, b_2\}$ has type $\{b_1+, b_2+, \varnothing\}$, while $\{\overline{b_1, b_2}\}$ has type $\{b_1-, b_2-, \omega\}$. The use of element and set variables γ and β allows for fine-grained polymorphism over set types. An atomic subtype ordering is also established, where a basic relation $+ \leq -$ is extended pointwise and covariantly over row and set types, so that, e.g., $\{b_1+, b_2+, \varnothing\} \leq \{b_1+, b_2-, \varnothing\}$, and contravariantly (resp. covariantly) in the domain (resp. range) type of functions. The subtyping relation mainly allows any set B to be used as a set $B' \subseteq B$ at points where at least B' is expected; record modification is dealt with (indeed, is the intended purpose of) row polymorphism, rather than subtyping.

Syntactic type safety for pml_B is easily established in the HM(X) framework [26]. By virtue of this property and Theorem 1, a sound, indirect static analysis for pop is immediately obtained by composition of the pop–to–pml_B transformation and pml_B type judgements:

Theorem 2 (Indirect type safety). *If e is a closed* pop *expression and $C, \Gamma \vdash [\![e]\!]_{d_1} : \sigma$ is valid, then e does not go wrong.*

The HM(X) framework provides the full generality of a constraint-based type system, which allows a unification-based approach to typing as a special case. In terms of logical type judgements, this entails taking C to always be the trivial **true** constraint. To take advantage of the readability of our type terms, we will henceforth consider a constraint-free approach, and let $\Gamma \vdash e : \tau$ denote $\mathbf{true}, \Gamma \vdash e : \tau$, keeping in mind that atomic subtyping is still available.

5.2 Direct types and subtyping

While the indirect type system described above is a sound static analysis for pop, it is desirable to define a direct static analysis for pop. The term transformation required for the indirect analysis is an unwanted complication for compilation, the indirect type system is not a clear declaration of program properties for the programmer, and type error reporting would be extremely troublesome. Thus, we define a direct type system for pop, the development of which significantly benefits from the transformational approach. In particular, type safety for the direct system may be demonstrated by a simple appeal to safety in the indirect system, rather than ab initio.

The direct type language for pop is defined in Fig. 10. We again ensure the construction of only meaningful types via kinding rules, defined in Fig. 11, hereafter considering only well-kinded pop types. The most novel feature of the pop type language is the form of object types $[\tau]^{\{\tau_1\}}_{\{\tau_2\}}$, where τ_2 is the type of any weakening set imposed on the object and τ_1 is the type of its interface. Types of sets are essentially the sets themselves, modulo polymorphic features; we abbreviate a type of the form $\tau; \epsilon$ or τ, ϵ as τ.

The close correlation between the direct and indirect type system begins with the type language: types for pop have a straightforward interpretation as pml_B types, defined in Fig. 12. This interpretation is extended to constraints and typing environments in the obvious manner. In this interpretation, we turn weakening sets "inside-out", in keeping with the manner in which weakening sets are turned inside-out in the pop-to-pml_B term transformation. The benefit of this approach is with regard to subtyping: weakening sets can be safely strengthened, and user interfaces safely weakened, in a uniform manner via pop subtyping, where we define $\tau \leq \tau'$ iff $(\!|\tau|\!) \leq (\!|\tau'|\!)$ in the pml_B type system. Recalling that in pml_B, subtyping mainly allows any set B to be used as a set $B' \subseteq B$ at points where at least B' is expected, our interpretation ensures that any object can be interpreted as one with *less* interface authorizations, e.g:

$$[m_1 : \tau_1; m_2 : \tau_2]^{\{d:\{m_1,m_2\}\}}_{\{\epsilon\}} \leq [m_1 : \tau_1; m_2 : \tau_2]^{\{d:\{m_1\}\}}_{\{\epsilon\}}\,.$$

Whereas since weakening sets are turned "inside-out" in the interpretation, subtyping allows any object to be interpreted as one with *greater* weaknesses:

$$[m_1 : \tau_1; m_2 : \tau_2]^{\{d:\{m_1,m_2\}\}}_{\{\epsilon\}} \leq [m_1 : \tau_1; m_2 : \tau_2]^{\{d:\{m_1,m_2\}\}}_{\{m_2\}}\,.$$

Note that since weakening is naturally expressed via subtyping, the type rule relevant to weakening (Weak) need

$$\begin{aligned}\tau ::= {} & \alpha, \beta, \ldots \mid \{\tau\} \mid [\tau] \mid [\tau]^\tau_\tau \mid m : \tau \to \tau\,;\ \tau \mid \\ & d : \tau\,;\ \tau \mid m, \tau \mid \epsilon\end{aligned}$$

Fig. 10. Direct pop type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \epsilon : \mathit{Meth}_M, \mathit{Set}_M, \mathit{Ifc}_D \qquad \frac{m \notin M \qquad \tau : \mathit{Set}_{M\cup\{m\}}}{m, \tau : \mathit{Set}_M} \qquad \frac{\tau_1 : \mathit{Type} \qquad \tau_2 : \mathit{Type} \qquad m \notin M \qquad \tau : \mathit{Meth}_{M\cup\{m\}}}{m : \tau_1 \to \tau_2\,;\ \tau : \mathit{Meth}_M}$$

$$\frac{\tau : \mathit{Meth}_\varnothing}{[\tau] : \mathit{Type}} \qquad \frac{\tau_1 : \mathit{Set}_\varnothing \qquad d \notin D \qquad \tau : \mathit{Ifc}_{D\cup\{d\}}}{d : \{\tau_1\}\,;\ \tau_2 : \mathit{Ifc}_D} \qquad \frac{\tau_1 : \mathit{Meth}_\varnothing \qquad \tau_2 : \mathit{Ifc}_\varnothing \qquad \tau_3 : \mathit{Set}_\varnothing}{[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} : \mathit{Type}}$$

Fig. 11. Direct pop type kinding rules

$$\begin{aligned}
(\!|\,\alpha\,|\!) &= \alpha \\
(\!|\,\tau\ \mathsf{ref}\,|\!) &= (\!|\,\tau\,|\!)\ \mathsf{ref} \\
(\!|\,\mathbf{int}\,|\!) &= \{\mathsf{obj} : \mathsf{int}\,;\ \mathsf{ifc} : \varnothing\,;\ \mathsf{strong} = \{\omega\}\} \\
(\!|\,[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}}\,|\!) &= \{\mathsf{obj} : \{(\!|\,\tau_1\,|\!)\}\,;\ \mathsf{ifc} : \{(\!|\,\tau_2\,|\!)\}\,;\ \mathsf{strong} : \{(\!|\,\tau_3\,|\!)_-\}\} \\
(\!|\,m : \tau_1 \to \tau_2\,;\ \tau\,|\!) &= m : (\!|\,\tau_1\,|\!) \to (\!|\,\tau_2\,|\!)\,;\ (\!|\,\tau\,|\!) \\
(\!|\,d : \{\tau_1\}\,;\ \tau_2\,|\!) &= d : \{(\!|\,\tau_1\,|\!)_+\}\,;\ (\!|\,\tau_2\,|\!) \\
(\!|\,_ : \{\tau\}\,|\!) &= \partial\{(\!|\,\tau\,|\!)_+\} \\
(\!|\,\epsilon\,|\!) &= \partial\{\varnothing\} \\
(\!|\,m, \tau\,|\!)_+ &= m+, (\!|\,\tau\,|\!)_+ \\
(\!|\,\epsilon\,|\!)_+ &= \varnothing \\
(\!|\,m, \tau\,|\!)_- &= m-, (\!|\,\tau\,|\!)_- \\
(\!|\,\epsilon\,|\!)_- &= \omega
\end{aligned}$$

Fig. 12. The pop-to-pml$_B$ type transformation

only force the appropriate subtyping coercion in derivations. An important feature of subtyping is that it allows only more constrained object usage, never increased authorizations.

The direct type judgement system for pop, the rules for which are *derived* from pml$_B$ type judgements for transformed terms, is defined in Fig. 13. The following definition simplifies the statement of the SEND rule:

Definition 7. *The relation $m \notin \tau_w$ holds iff $\neg\exists\beta.((\!|\,\tau_w\,|\!)_+ \leq (\!|\,m, \beta\,|\!)_+)$, where $\beta \notin fv(\tau_w)$.*

The easily proven, tight correlation between the indirect and direct pop type systems is clearly demonstrated via the following lemma; the proof is straightforward, since the direct type judgements can be viewed simply as syntactic sugar for pml$_B$ type judgements:

Lemma 13. *$d, \Gamma \vdash e : \tau$ is valid iff $(\!|\,\Gamma\,|\!) \vdash [\![e]\!]_d : (\!|\,\tau\,|\!)$ is valid.*

And in fact, along with Theorem 1, this correlation is sufficient to establish direct type safety for pop:

Theorem 3 (Direct type safety). *If e is a closed* pop *expression and $d, \Gamma \vdash e : \tau$ is valid, then e does not go wrong.*

This result demonstrates the advantages of the transformational method, which has allowed us to define a direct, expressive static analysis for pop with a minimum of proof effort.

5.3 Optimizations

Another benefit of our static analysis for pop is that security checks in well-typed programs may be eliminated at run time, since well-typed programs are guaranteed to be safe. The optimizations that can be effected are quite substantial, as illustrated by an optimized semantics for pop defined in Fig. 14. Since well-typing rules out programs that go wrong, due to security errors or otherwise, run-time security checks and manipulation of interfaces are eliminated in the optimized semantics. (Note that weakening amd casting become no-ops.) Domain stacks are also safely eliminated, as is the associated expression framing. For well-typed expressions, this optimized semantics is equivalent to the nonoptimized semantics of Sect. 3.2, modulo security annotations on objects. That is, letting $\simeq$ be the same as syntactic equivalence for pop expression forms except for objects, where $(co \cdot \varphi\backslash_\iota) \simeq (co \cdot \varphi'\backslash_{\iota'})$, we easily obtain the following result by definitions of $\to$, $\rightsquigarrow$, and Theorem 3:

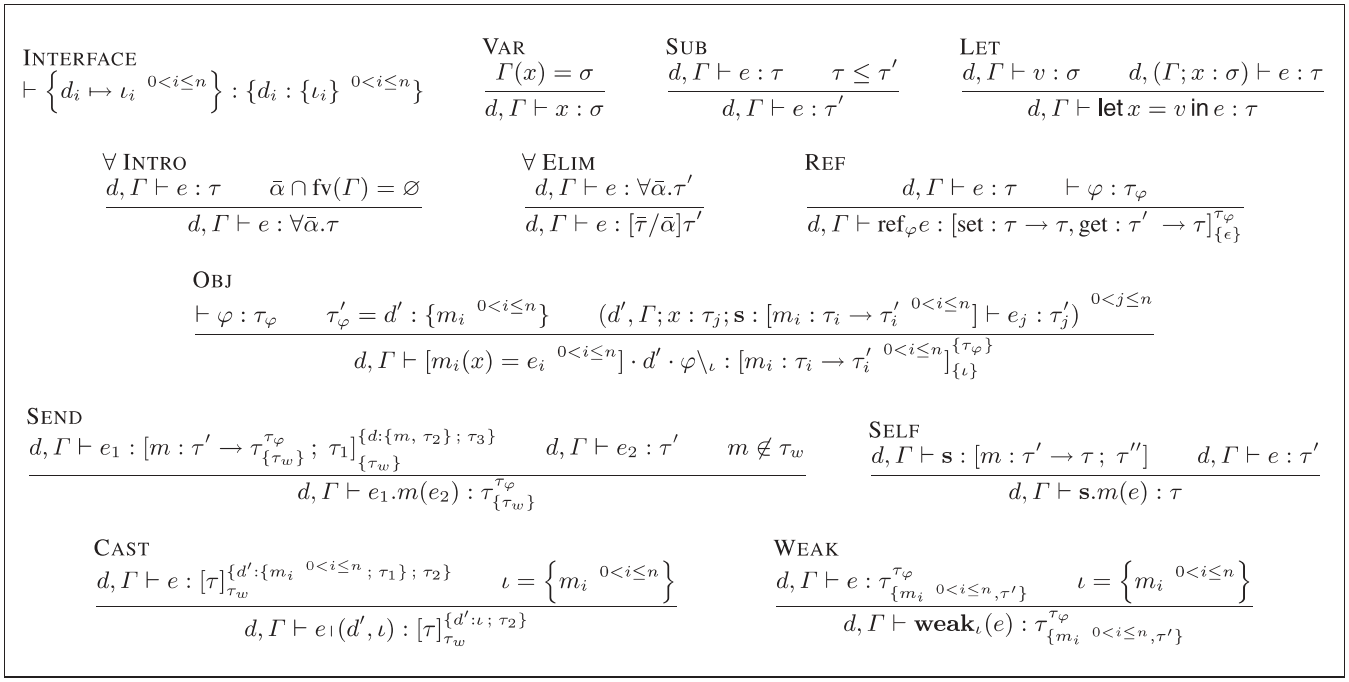

Fig. 13. Direct type judgements for pop

Corollary 2. *If e is a well-typed top-level* pop *expression, then e evaluates to v in the nonoptimized semantics iff e evaluates to v' in the optimized semantics such that $v \simeq v'$.*

Conceivably, the semantics can be optimized even further; since all run-time security checks can be eliminated, weakening and casting operations, as well as weakening, interface, and domain annotations on objects, can be erased from the source code after typing. Here we have only suggested the possible extent of optimizations and shown how such optimizations are easily shown to be safe for evaluation of well-typed expressions.

5.4 Type inference

The transformational method allows a similarly simplified approach to the development of type inference. The HM(X) framework comes equipped with a type inference algorithm modulo a constraint normalization procedure (constraint normalization is the same as constraint satisfaction, e.g., *unification* is a normalization procedure for equality constraints). Furthermore, efficient constraint normalization procedures have been developed for row types [18, 22], and even though set types are novel, their interpretation as row types [27] allows a uniform implementation. This yields a type inference algorithm for pml_B in the HM(X) framework. An indirect inference analysis for pop may then be im-

$$
\begin{array}{rll}
([\varrho] \cdot d' \cdot \varphi\backslash_\iota).m(v), \sigma \rightsquigarrow e[[\varrho]/\mathbf{s}][v/x], \sigma & \text{if } (m(x) = e) \in \varrho & (\textit{send}) \\
[\varrho].m(v), \sigma \rightsquigarrow e[[\varrho]/\mathbf{s}][v/x], \sigma & \text{if } (m(x) = e) \in \varrho & (\textit{self}) \\
o\,|(d', \iota'), \sigma \rightsquigarrow o, \sigma & & (\textit{cast}) \\
\mathbf{weak}_\iota(o), \sigma \rightsquigarrow o, \sigma & & (\textit{weaken}) \\
\mathrm{ref}_\varphi v, \sigma \rightsquigarrow l \cdot \varphi\backslash_\varnothing, \sigma[l \mapsto v] & \text{if } l \notin \mathrm{dom}(\sigma) & (\textit{newcell}) \\
(l \cdot \varphi\backslash_\iota).\mathrm{set}(v), \sigma \rightsquigarrow v, \sigma[l \mapsto v] & \text{if } l \in \mathrm{dom}(\sigma) & (\textit{set}) \\
(l \cdot \varphi\backslash_\iota).\mathrm{get}(), \sigma \rightsquigarrow \sigma(l), \sigma & & (\textit{get}) \\
\mathsf{let}\, x = v \,\mathsf{in}\, e, \sigma \rightsquigarrow e[v/x], \sigma & & (\textit{let}) \\
E[e], \sigma \rightsquigarrow E[e'], \sigma' & \text{if } e, \sigma \rightsquigarrow e', \sigma' & (\textit{context})
\end{array}
$$

Fig. 14. Optimized operational semantics for pop

mediately obtained as the composition of the pop-to-pml_B transformation and pml_B type inference. This analysis is potentially constraint free; simply disallowing recursive constraints and interpreting constraint normalization as satisfaction yields an inference algorithm that implements the constraint-free logical type system for pml_B.

Beyond this, a direct type inference algorithm can be derived from the indirect algorithm, just as direct type judgements can be derived from indirect judgements. Only the pop syntactic cases need be adopted since constraint satisfaction procedures for row types may be reused in this context. (Recall that the direct pop type language has a simple interpretation in the pml_B type language.)

6 Using pop

In this section we provide several examples that demonstrate the usage and flexibility of the pop system, including a scheme for embedding the ownership types of [9] in pop in a type-safe manner, as well as a scheme for encoding class definitions with `public`, `private`, and `protected` instance modifiers.

6.1 Basic typing examples

Here is a brief example illustrating the features of pop and the expressiveness of its direct type system. We may create a cell c that is read-write in domain d but read-only elsewhere, containing a value v, as follows:

$$c = \text{ref}_{\{d\mapsto\{\text{get,set}\},\partial\mapsto\{\text{get}\}}}(v)\,.$$

Then supposing $v : \tau$, the cell c has the following type:

$$c : [\text{get} : \text{unit} \to \tau, \text{set} : \tau \to \tau]^{\{d:\{\text{get,set}\},\partial:\{\text{get}\}}_{\{\}}\,.$$

Note how the interface is expressed in the type and how no weakenings show up in the type. However, if we read-weaken c, this information is expressed in the type:

$$\mathbf{weak}_{\{\text{set}\}}(c) : [\text{get} : \text{unit} \to \tau, \text{set} : \tau \to \tau]^{\{d:\{\text{get,set}\},\partial:\{\text{get}\}}_{\{\text{set}\}}\,.$$

Given the requirements of the SEND rule, attempting to use the set method of this weakened capability will not be well-typed in any context, nor will an attempted set of v returned by reading the weakened capability. This is true even assuming that v is a cell, since weakening information is propagated to τ by the type system, just as weakening is propagated to v by the operational semantics:

$$(\mathbf{weak}_{\{\text{set}\}}(c)).\text{set}(e) \quad \textit{not well-typed}$$

$$\mathsf{let}\, c' = (\mathbf{weak}_{\{\text{set}\}}(c)).\text{get}()\,\mathsf{in}\, c'.\text{set}(e) \quad \textit{not well-typed}.$$

6.2 Ownership types

A language model for alias analysis, together with an "ownership type" analysis, is proposed in [7–9]. At the heart of this work is the so-called containment relation and invariant, which mediates references between objects in a principled manner. Here we show how this basic containment relation and invariant can be captured in pop by choosing an appropriate naming scheme (albeit with a use-based security model rather than the communication-based model of [7–9]). We note that there are extensions of the ownership types system introduced in [7, 8] that cannot be captured in pop, e.g., "owner polymorphism", but this encoding does model the basic calculus [9].

Assume the following object definition in the language of [9], with the containment relation $p_1 \prec: p_2 \prec: p_3$:

$$[m(x) = e]^{p_2}_{p_1}.$$

A similar specification can be defined and statically enforced in pop with the following object definition:

$$[m(x) = e] \cdot p_1 \cdot \{p_1 \mapsto \{m\}\,, p_2 \mapsto \{m\}\,, p_3 \mapsto \{m\}\}\ .$$

In general, given any set of contexts $\mathcal{C}$, partial ordering $(\mathcal{C}, \prec:)$, and object o^p_q, we can transform the object into the form $o \cdot q \cdot \varphi$, where $\text{dom}(\varphi) = \{p' \mid p' \prec: p\}$ and for all $p \in \text{dom}(\varphi)$, $\varphi(p)$ is all of o's methods, and carry the transformation recursively through any objects defined in o's methods. In this context, well-typing is analogous to the static enforcement of the containment invariant.

6.3 Classes, `private` *and* `protected`

By choosing different naming schemes, a variety of security paradigms can be effectively and reliably expressed in pop. One such scheme enforces a strengthened meaning of the `private` and `protected` modifiers in class definitions, a focus of other communication-based capability type analyses [9, 29]. As demonstrated in [29], a `private` field can leak by being returned by reference from a `public` method. Here we show how this problem can be addressed in a use-based model. Assume the following Java-like pseudocode package p, containing class definitions c_1, c_2, and possibly others, where c_2 specifies a method m that leaks a `private` instance variable:

```
package p begin

class c1 {                    class c2 {
  public:                       public:
    f(x) = x                      m(x) = b
  private:                        a = new c1
    g(x) = x                    private:              ...
  protected:                      b = new c1
    h(x) = x                    protected:
}                                 c = new c1
                              }

end
```

We can implement this definition as follows. Interpreting domains as class names in pop, let $c_1, \ldots, c_n$ be the names of classes in package p. Then, the appropriate interface for objects in the encoding of class c_1 is as follows:

$$\varphi_1 \triangleq \{c_1 \mapsto \{f, g, h\}, c_2 \mapsto \{f, h\}, \ldots, c_n \mapsto \{f, h\}, \\ \partial \mapsto \{f\}\} .$$

The class c_1 can then be encoded as an object *factory*, an object with only one publicly available method that returns new objects in the class, and some arbitrary label d:

$$o_1 \triangleq [f(x) = x,\ g(x) = x,\ h(x) = x] \cdot c_1 \cdot \varphi_1$$
$$\text{fctry}_{c_1} \triangleq [\text{new}(x) = o_1] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\} .$$

To encode c_2, we again begin with the obvious interface definition for objects in the encoding of class c_2, letting $\iota = \{m, a, c\}$:

$$\varphi_2 \triangleq \{c_2 \mapsto \{m, a, b, c\}, c_1 \mapsto \iota, c_3 \mapsto \iota, \ldots, c_n \mapsto \iota, \\ \partial \mapsto \{m, a\}\} .$$

However, we must now encode *instance variables*, in addition to methods. In general, this is accomplished by encoding instance variables a containing objects as methods $a()$ that return references to objects. Then, any selection of a is encoded as $a().\text{get}()$, and any update with v is encoded $a().\text{set}(v)$. By properly constraining the interfaces on these references, a "Java level" of modifier enforcement can be achieved; but casting the interfaces of stored objects *extends* the security by making objects *unusable* outside the intended domain. Let $e\,|\,(\{d_1, \ldots, d_n\}, \iota)$ be sugar for $e\,|\,(d_1, \iota)\,|\cdots|\,(d_n, \iota)$. Using fctry_{c_1}, we may create a `public` version of an object equivalent to o_1, without any additional constraints on its confinement, as follows:

$$o_a \triangleq \text{fctry}_{c_1}.\text{new}() .$$

Letting $p' = p - \{c_2\}$, we may create a version of an object equivalent to o that is `private` with respect to the encoding of class c_2, using casts as follows:

$$o_b \triangleq (\text{fctry}_{c_1}.\text{new}())\,|\,(\partial, \varnothing)\,|\,(p', \varnothing) .$$

We may create a version of an object equivalent to o that is `protected` with respect to the encoding of package p, as follows:

$$o_c \triangleq (\text{fctry}_{c_1}.\text{new}())\,|\,(\partial, \varnothing) .$$

Let o_2 be defined as follows:

$$o_2 \triangleq \mathsf{let}\, r_a = \text{ref}_{\{\partial \mapsto \{\text{set},\text{get}\}\}} o_a \,\mathsf{in} \\ \mathsf{let}\, r_b = \text{ref}_{\{c_1 \mapsto \{\text{set},\text{get}\}\}} o_b \,\mathsf{in} \\ \mathsf{let}\, r_c = \text{ref}_{\{c_1 \mapsto \{\text{set},\text{get}\}, p \mapsto \{\text{set},\text{get}\}\}} o_c \,\mathsf{in} \\ [\, m(x) = \mathbf{s}.b().\text{get}(), \\ a(x) = r_a, \\ b(x) = r_b, \\ c(x) = r_c \,] \cdot c_2 \cdot \varphi_2 .$$

Then fctry_{c_2} is encoded, similarly to fctry_{c_1}, as:

$$\text{fctry}_{c_2} \triangleq [\text{new}(x) = o_2] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\} .$$

Given this encoding, if an object stored in b is leaked by a nonlocal use of m, it is unusable. This is the case because, even though a nonlocal use of m will return b, in the encoding this return value explicitly states it cannot be used outside the confines of c_2; as a result of the definition of φ_1 and casting, the avatar o_b of b in the encoding has an interface equivalent to:

$$\{c_2 \mapsto \{f, h\}, p' \mapsto \varnothing, \partial \mapsto \varnothing\} .$$

While the communication-based approach accomplishes a similar strengthening of modifier security, the benefits of greater flexibility may be enjoyed via the use-based approach. For example, a `protected` reference can be safely passed outside of a package and then back in, as long as a use of it is not attempted outside the package. Also for example are the fine-grained interface specifications allowed by this approach, enabling greater modifier expressivity, e.g., publicly read-only but privately read/write instance variables.

7 Conclusion

As shown in [2], object confinement is an essential aspect of securing OO programming languages. Related work on this topic includes the *confinement types* of [29], which have been implemented as an extension to Java [6]. The mechanism is simple: classes marked **confined** must not have references escape their defining package. Most closely related are the *ownership types* of [9], discussed in Sect. 6.2. However, as discussed in Sect. 2, these previous type approaches treat a communication-based mechanism, while one of the main points of this paper is the importance of studying the use-based approach as an alternative. Furthermore, our type system is polymorphic, with inference methods readily available due to its basis in row types.

Topics for future work include an extension of the language to capture *inheritance*, an important OO feature that presents challenges for type analysis. Also, we hope to study capability *revocation*.

In summary, contributions of this work include a focus on the more expressive use-based security model, the first type-based characterization of weak capabilities, and a general mechanism for fine-grained, use-based security specifications that includes flexible domain naming, precise object interface definitions, and domain-specific interface casting. Furthermore, we have defined a static

analysis that enforces the security model, with features including flexibility due to polymorphism and subtyping, declarative benefits due to readability, and ease of proof due to the use of transformational techniques.

References

1. Aldrich J, Kostadinov V, Chambers C (2002) Alias annotations for program understanding. In: Proceedings of the 17th ACM conference on object-oriented programming, systems, languages, and applications. ACM Press, New York, pp 311–330
2. Banerjee A, Naumann D (2002) Representation independence, confinement and access control. In: Conference Record of POPL02: The 29TH ACM SIGPLAN-SIGACT symposium on principles of programming languages, Portland, OR, January 2002, pp 166–177
3. Banerjee A, Naumann D (2003) Using access control for secure information flow in a java-like language. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW03)
4. Bokowski B, Vitek J (1999) Confined types. In: Proceedings of the 14th annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications (OOPSLA), November 1999
5. Boyland J, Noble J, Retert W (2001) Capabilities for aliasing: a generalisation of uniqueness and read-only. In: ECOOP'01 – Object-oriented programming, 15th European conference. Lecture notes in computer science, vol 2072. Springer, Berlin Heidelberg New York
6. Bryce C, Vitek J (1999) The JavaSeal mobile agent kernel. In: 1st international symposium on agent systems and applications (ASA'99)/3rd international symposium on mobile agents (MA'99), Palm Springs, CA
7. Clarke D (2001) An object calculus with ownership and containment. In: FOOL8 – the 8th international workshop on foundations of object-oriented languages
8. Clarke D, Drossopoulou S (2002) Ownership, encapsulation and the disjointness of type and effect. In: Conference on object-oriented programming systems, languages and applications (OOPSLA)
9. Clarke D, Noble J, Potter J (2001) Simple ownership types for object containment. In: ECOOP'01 – Object-oriented programming, 15th European conference. Lecture notes in computer science, vol 2072. Springer, Berlin, Heidelberg, New York
10. Miller M, et al The E programming language. `http://www.erights.org`
11. Fournet C, Gordon AD (2002) Stack inspection: theory and variants. In: Proceedings of the 29th symposium on principles of programming languages (POPL'02), January 2002
12. Hawblitzel C, Chang C-C, Czajkowski G, Hu D, von Eicken T (1998) Implementing multiple protection domains in Java. In: 1998 USENIX annual technical conference, New Orleans, pp 259–270
13. Hennessy M, Riely J (2002) Resource access control in systems of mobile agents. Inf Comput 173:83–120
14. Kain RY, Landwehr CE (1987) On access checking in capability-based systems. IEEE Trans Softw Eng 13(2): 202–207
15. Leino KRM, Nelson G (2002) Data abstraction and information hiding. ACM Trans Programm Lang Syst 24(5): 491–553
16. Müller P, Poetzsch-Heffter A (1999) Universes: a type system for controlling representation exposure. In: Poetzsch-Heffter A, Meyer J (eds) Programming languages and fundamentals of programming, Technical Report, vol 263. Fernuniversität Hagen
17. Odersky M, Sulzmann M, Wehr M (1999) Type inference with constrained types. Theory Practice Object Syst 5(1):35–55
18. Pottier F (2000) A versatile constraint-based type inference system. Nordic J Comput 7(4):312–347
19. Pottier F, Conchon S (2000) Information flow inference for free. In: Proceedings of the the 5th ACM SIGPLAN international conference on functional programming (ICFP'00), September 2000, pp 46–57
20. Pottier F, Skalka C, Smith S (2001) A systematic approach to static access control. In: Sands D (ed) Proceedings of the 10th European symposium on programming (ESOP'01), April 2001. Lecture notes in computer science, vol 2028. Springer, Berlin Heidelberg New York, pp 30–45
21. Rémy D (1992) Projective ML. In: 1992 ACM conference on Lisp and functional programming, New York. ACM Press, New York, pp 66–75
22. Rémy D (1993) Syntactic theories and the algebra of record terms. Research Report 1869, INRIA
23. Rémy D (1993) Typing record concatenation for free. In: Gunter CA, Mitchell JC (eds) Theoretical aspects of object-oriented programming: types, semantics and language design. MIT Press, Cambridge, MA
24. Shapiro J, Weber S (2000) Verifying the EROS confinement mechanism. In: 21st IEEE symposium on research in security and privacy
25. Skalka C (2002) Types for programming language-based security. PhD thesis, Johns Hopkins University, Baltimore, MD
26. Skalka C, Pottier F (2003) Syntactic type soundness for HM(X). Electronic notes in theoretical computer science, vol 75
27. Skalka C, Smith S (2003) Set types and applications. Electronic notes in theoretical computer science, vol 75
28. van Doorn L, Abadi M, Burrows M, Wobber E (1996) Secure network objects. In: IEEE symposium on security and privacy, May 1996
29. Vitek J, Bokowski B (2001) Confined types in java. Softw Practice Exper 31(6):507–532
30. Walker D (2000) A type system for expressive security policies. In: Conference record of POPL'00: The 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages, Boston, MA, January 2000, pp 254–267