Electronic Notes in Theoretical Computer Science 75 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume75.html 20 pages

Set Types and Applications

Christian Skalka [1]

The University of Vermont

Scott Smith [2]

The Johns Hopkins University

Abstract

We present pml_B, a programming language that includes primitive sets and associated operations. The language is equipped with a precise type discipline that statically captures dynamic properties of sets, allowing runtime optimizations. We demonstrate the utility of pml_B by showing how it can serve as a practical *implementation* language for higher-level programming language based security systems, and characterize pml_B by comparing the expressiveness of pml_B sets with enumerations.

1 Introduction

In this paper we present the pml_B programming language, which includes primitive records, sets, and associated operations, as well as a static type discipline that provides accurate specifications of these constructs. We demonstrate how pml_B can serve as a practical *implementation* language for higher-level language features that can be encoded in terms of sets and records; in particular, the type discipline of pml_B allows dynamic set manipulation and membership checks to be captured statically, which in turn allows runtime optimizations. There are a wide array of set-based properties in languages studied today, so there are multiple applications of pml_B.

The pml_B language of records includes default values in the style of Rémy's Projective ML [12]. The language of sets includes syntax for defining sets of atomic elements, as well as operations such as intersection, union, difference, etc. Sets are at first approximation records, where all values are of trivial type `unit`. However, since sets are simpler than records, there are set operations which can be effectively modeled statically that are difficult or impossible in the case of records, and set types can also be simpler than record types. We

[1] Email: `skalka@cs.uvm.edu`
[2] Email: `scott@cs.jhu.edu`

equip the language with a type system that accurately specifies the contents of records, sets, and the results of associated operations; we also show that this type system is sound. To define the type system, we make use of the HM(X) framework [16], instantiating it with a constraint system containing *row types* [13] and *conditional constraints* [9]; row types were originally developed for application to records with default values; we show here how they can also be used to type sets which include new operations not defined for record row types [10,17].

The language and type system presented here is a more general version of the λ_{set} language and type system, defined in [10]. In that presentation, one element of set union and intersection operations was always statically known; in pml_B, these operations are fully generalized. The pml_B language also extends λ_{set} with a general set difference operation, cofinite set definitions, and extensible records. Thus, the language is significantly more expressive. Also more expressive is the interpretation of conditional constraints, which provides accurate types for pml_B set operations, and we define an abbreviated type form for more succinct and readable set types.

The pml_B language is most useful as an implementation language for higher-level languages, its type system serving as an indirect static analysis and specification of high-level features, allowing run-time checks to be eliminated as a result of type safety. Thus, the pml_B language facilitates run-time *optimizations* of source languages. We discuss examples of this, including an implementation of Java-style stack inspection [10], and an implementation of an Object-Oriented language model with object confinement mechanisms [17]. Using pml_B to implement these languages has distinct technical benefits; in particular, type safety in the source languages is easily developed on the uniform foundation of pml_B type safety.

To characterize the expressiveness of the pml_B language, and to suggest alternate implementations of the semantics and type system, we also present some observations on the duality of set types and enumeration types. In particular, we show that sets can be fully and faithfully encoded with enumerations, and enumerations can be fully and faithfully encoded with records.

2 The pml_B language: syntax and semantics

The grammar for pml_B is given in Fig. 1, the semantics in Fig. 2. The language is based on Rémy's Projective ML [12], containing records with default values, manipulated with the *elevation* and *modification* record constructors $\{e\}$ and $e\{a = e'\}$, and the *projection* destructor $e.a$. For example, $\{1\}\{a_1 = \text{true}\}\{a_2 = 2.1\}$ is a record with the a_1 field set to true and the a_2 field set to 2.1, and with default contents 1– that is, *every* other field (a countably infinite collection) is implicitly set to 1.

We take as given a countably infinite set of record labels $\mathcal{L}_a$, and a countably infinite set of atomic set elements $\mathcal{L}_b$. Borrowing language from set

$$x \in \mathcal{V},\ a \in \mathcal{L}_a,\ b \in \mathcal{L}_b,\ A \subseteq \mathcal{L}_a,\ B \subseteq \mathcal{L}_b \qquad \textit{identifiers}$$

$$v ::= \mathsf{fix}\, z.\lambda x.e \mid s \mid \{v\} \mid v\{a = v\} \qquad \textit{values}$$

$$s ::= B \mid \bar{B} \mid \vee \mid \wedge \mid \ominus \mid \ni_b \mid ?_b \qquad \textit{sets, set operations}$$

$$e ::= x \mid v \mid e\, e \mid \mathsf{let}\, x = v\, \mathsf{in}\, e \mid \{e\} \mid e\{a = e\} \mid e.a \qquad \textit{expressions}$$

$$E ::= [\,] \mid E\, e \mid v\, E \mid \{E\} \mid E\{a = e\} \mid v\{a = E\} \mid E.a \qquad \textit{eval. contexts}$$

Fig. 1. Grammar for pml_B

theory, we refer to the latter as *urelements*. The language allows definition of finite sets B of urelements $b \in \mathcal{L}_b$, and countably infinite cosets $\bar{B}$. This latter feature presents some practical implementation issues, but in this presentation we take it at mathematical "face value"— that is, we take $\bar{B}$ to denote $\mathcal{L}_b \backslash_B$. Basic set operations are provided, including $\ni_b$, $\wedge$, $\vee$ and $\ominus$, which are membership check, intersection, union and difference operations, respectively. Also provided is a set membership test operation $?_b$, which allows branching on the presence or absence of a set element in a given set, as opposed to failure in the case of absence à la $\ni_b$. For clarity of presentation, we define the following syntactic sugar:

$$(\ni_b e) \triangleq (e \ni b)$$
$$(\wedge e_1 e_2) \triangleq (e_1 \wedge e_2)$$
$$(\vee e_1 e_2) \triangleq (e_1 \vee e_2)$$
$$(\ominus e_1 e_2) \triangleq (e_1 \ominus e_2)$$

To abbreviate certain function definitions, we take $\lambda x.e$ to denote the function $\mathsf{fix}\, z.\lambda x.e$ where z does not occur free in e.

3 The type constraint system RS

We define the type system for pml_B as an instance of the HM(X) framework [16,7]. The HM(X) framework provides a functional language core and type system with let-polymorphism; the type judgement rules for this functional core are defined in Fig. 3, where constrained type schemes $\forall\bar{\alpha}[C].\tau$ are denoted σ, and constraint entailment (resp. scheme consistency) is denoted $C \Vdash D$ (resp. $C \Vdash \sigma$) and is defined near the end of Sect. 3.3. This core can be specialized by instantiation with a *sound* type constraint system, and by extension with additional language constants and their initial type bindings, which must be sound with respect to their semantics (the so-called *δ-typability*

$$
\begin{array}{rcll}
(\mathsf{fix}\, z.\lambda x.e)v & \to & e[v/x][\mathsf{fix}\, z.\lambda x.e/z] & & (\beta) \\
\mathsf{let}\, x = v \,\mathsf{in}\, e & \to & e[v/x] & & (let) \\
\{v\}.a & \to & v & & (default) \\
v_1\{a = v_2\}.a & \to & v_2 & & (access) \\
v_1\{a' = v_2\}.a & \to & v_1.a & a' \neq a & (skip) \\
B \ni b & \to & B & \text{if } b \in B & (memcheck) \\
B_1 \wedge B_2 & \to & B_1 \cap B_2 & & (intersect) \\
B_1 \vee B_2 & \to & B_1 \cup B_2 & & (union) \\
B_1 \ominus B_2 & \to & B_1 - B_2 & & (difference) \\
?_b\, B & \to & \lambda f.\lambda g.f(B) & \text{if } b \in B & (memtesty) \\
?_b\, B & \to & \lambda f.\lambda g.g(B) & \text{if } b \notin B & (memtestn) \\
E[e] & \to & E[e'] & \text{if } e \to e' & (context)
\end{array}
$$

Fig. 2. Operational semantics for pml_B

property). Any specialization meeting these requirements enjoys type soundness in the framework. The details of HM(X) are omitted here for brevity; interested readers are referred to the previous citations.

The type analysis for pml_B is defined, in part, by instantiating HM(X) with the RS type constraint system, comprising row types and conditional constraints. The definition includes the type and constraint language itself (Sect. 3.1), together with its logical interpretation in a model (Sect. 3.2 and Sect. 3.3).

3.1 The type and constraint language

The syntax of types and constraints is defined in Fig. 4, where ℓ ranges over $\mathcal{L}_a \cup \mathcal{L}_b$. The syntax contains language for expressing *record* and *set* types (hence the name RS: Records and Sets).

To describe the contents of sets and records, we use *polymorphic row types* [12,13], which were originally proposed to describe extensible records with default contents. Recalling the example record $\{1\}\{a_1 = \mathrm{true}\}\{a_2 = 2.1\}$ from Sect. 2, in the rows setting a type of this record is $\{a_1 : \mathrm{bool}\,;\ a_2 : \mathrm{real}\,;\ \partial\mathrm{int}\}$, where the row type ∂int specifies that every field not otherwise mentioned in the type contains an int. Since in general the row type $\partial\tau$ is shorthand for an

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma \qquad C \Vdash \sigma}{C, \Gamma \vdash x : \sigma}$$

$$\textsc{Sub}\quad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}$$

$$\textsc{Abs}\quad \frac{C, (\Gamma; x : \tau) \vdash e : \tau'}{C, \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\textsc{App}\quad \frac{C, \Gamma \vdash e_1 : \tau_2 \to \tau \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\quad \frac{C, \Gamma \vdash e_1 : \sigma \qquad C, (\Gamma; x : \sigma) \vdash e_2 : \tau}{C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}$$

$$\forall\ \textsc{Intro}\quad \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau}$$

$$\forall\ \textsc{Elim}\quad \frac{C, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau' \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

Fig. 3. The system HM(X)

infinite, pointwise-uniform type specification, it can be unrolled, so that:

$$\partial\tau = (a_3 : \tau\,;\ \partial\tau) = (a_3 : \tau\,;\ a_4 : \tau\,;\ \partial\tau) = \cdots$$

Thus, another type for the previous example record is $\{a_1 : \mathrm{bool}\,;\ a_2 : \mathrm{real}\,;\ a_3 : \mathrm{int}\,;\ \partial\mathrm{int}\}$. The ability of row types to finitely specify infinite program objects makes them especially useful for describing the potentially cofinite sets of pml_B, and fine-grained polymorphism over row field types makes them useful for describing the set operations of pml_B. The original presentation of rows [12,13] also includes an equational theory, which in particular allow rows to commute (i.e., field orderings don't matter). Here these equations are not axiomatic, but rather they hold as a result of the interpretation defined in Sect. 3.3.

Record types are built up from row types ρ using the record type constructor $\{\rho\}$. *Set types* are also built up from a particular form of row types, using the set type constructor $\{\cdot\rho\cdot\}$. These particular row types are built up from *presence* constructors, which specify whether a given element may be present in a set ($+$), may not be present in it ($-$), may or may not appear in it ($\top$), or whether this information is irrelevant, because the set itself is unavailable ($\bot$) (*NB*: $\bot$ and $\top$ here are *not* the same as the "top" and "bottom" types in non-structural subtyping systems!). This form is enforced by the kinding rules, defined below. We will also define a succinct, more readable form of

$$
\begin{array}{rlr}
\tau ::= & \alpha, \beta, \ldots \mid \tau \to \tau \mid \{\tau\} \mid \{\cdot\tau\cdot\} \mid \ell : \tau \,;\, \tau \mid \partial\tau \mid c & \textit{types} \\
c ::= & + \mid - \mid \top \mid \bot & \textit{constructors} \\
C ::= & \mathbf{true} \mid C \wedge C \mid \exists\alpha.C \mid \tau = \tau \mid \tau \leq \tau \mid \text{if } c \leq \tau \text{ then } \tau \leq \tau & \textit{constraints}
\end{array}
$$

Fig. 4. RS Grammar

set types in Sect. 3.4, which are defined as syntactic sugar for the primitive form. A significant consequence of this primitive definition of set types as specialized rows, is that set types can be implemented by *re-use* of existing row type implementations.

The constraint language of RS offers standard equality and subtyping constraints, as well as conditional constraints. To ensure that only meaningful types and constraints can be built, we equip them with *kinds*, defined by:

$$k ::= \mathit{Con} \mid \mathit{Row}(\tau)_A \mid \mathit{Row}(c)_B \mid \mathit{Type}$$

where A ranges over finite subsets of field labels $\mathcal{L}_a$ and B ranges over finite subsets of set urelements $\mathcal{L}_b$. Row kinds are parameterized by τ or c, specifying whether they describe the contents of a record or a set, respectively. For every kind k, we assume given a distinct, denumerable set of *type variables* $\mathcal{V}_k$. We use $\alpha, \beta, \gamma, \ldots$ to represent type variables. From here on, we consider only *well-kinded* types and constraints, as defined in Fig. 5 and Fig. 6. The formal purpose of these rules is to guarantee that every constraint has an interpretation in our model, defined in Sect. 3.2. Intuitively, aside from the usual well-formedness properties of type terms, the type kinding rules guarantee that no row type mentions a particular field twice. The constraint kinding rules admit two syntactic forms of conditional constraints, one in which two type constructors are related in the condition, and one in which a type constructor c is related with a row τ in the condition. This latter constraint may seem ill-formed, but it turns out to be an abbreviation for an infinite collection of conditional constraints, conditioned on relations of the constructor c with every field type in τ. The behavior of conditional constraints is precisely specified by our interpretation of constraints, defined in Sect. 3.3 and discussed in Example 4.1.

3.2 The model

The model for RS is constructed by associating with every kind k a mathematical structure denoted $[\![k]\!]$. Informally, each of these structures contain inductively defined variable-free elements, which are either presence constructors, functions from field labels to other elements, or elements built up with the binary constructor $\to$ and the unary constructors $\{\cdot\}$ and $\{\cdots\}$. These

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

$$\frac{\tau : \mathit{Type}}{\partial\tau : \mathit{Row}(\tau)_A} \qquad \frac{\tau : \mathit{Con}}{\partial\tau : \mathit{Row}(c)_B} \qquad c : \mathit{Con}$$

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

$$\frac{\tau : \mathit{Type} \qquad a \notin A \qquad \tau' : \mathit{Row}(\tau)_{A \cup \{a\}}}{(a : \tau\, ;\ \tau') : \mathit{Row}(\tau)_A}$$

Fig. 5. Kinding rules for RS types

$$\frac{}{\vdash \mathbf{true}} \qquad \frac{\vdash C_1, C_2}{\vdash C_1 \wedge C_2} \qquad \frac{\vdash C}{\vdash \exists\alpha.C} \qquad \frac{\tau, \tau' : k}{\begin{array}{c}\vdash \tau = \tau' \\ \vdash \tau \le \tau'\end{array}}$$

$$\frac{\tau : \mathit{Con} \qquad \vdash C}{\vdash \text{if } c \le \tau \text{ then } C} \qquad \frac{\tau, \tau', \tau'' : \mathit{Row}(c)_B}{\vdash \text{if } c \le \tau \text{ then } \tau' \le \tau''}$$

Fig. 6. Kinding rules for RS constraints

structures are formally defined below. We let $\hat{\tau}$ range over elements of any kind. Each structure $[\![k]\!]$ is equipped with a *partial ordering* $\le$ of its elements. Accordingly, the relation on each $[\![k]\!]$ is defined to be transitive and reflexive, by requiring the following inferences to be axiomatic for all $\hat{\tau}$:

$$\hat{\tau} \le \hat{\tau} \qquad\qquad \frac{\hat{\tau} \le \hat{\tau}' \qquad \hat{\tau}' \le \hat{\tau}''}{\hat{\tau} \le \hat{\tau}''}$$

The model is explicated for each $[\![k]\!]$ as follows:

$[\![\mathit{Con}]\!]$: The elements of $[\![\mathit{Con}]\!]$ are contained in the set $\{+, -, \bot, \top\}$. As is made clear by the full definition of our model, continued below, the characteristics of the ordering $\le$ over the model is determined by the definition of $\le$ over $[\![\mathit{Con}]\!]$; if we define $\le$ over $[\![\mathit{Con}]\!]$ as equality, then $\le$ is an equivalence relation over the entire model– that is, over each $[\![k]\!]$. On the other hand, we may choose a subtype ordering over $[\![\mathit{Con}]\!]$, axiomatized as follows:

$$\bot \le + \qquad \bot \le - \qquad + \le \top \qquad - \le \top$$

$$\rho(\tau \to \tau') = \rho(\tau) \to \rho(\tau')$$
$$\rho(\{\tau\}) = \{\rho(\tau)\} \qquad \rho(\{\cdot\tau\cdot\}) = \{\cdot\rho(\tau)\cdot\}$$
$$\rho(\ell : \tau\,;\ \tau')(\ell) = \rho(\tau) \qquad \rho(\ell : \tau\,;\ \tau')(\ell') = \rho(\tau')(\ell') \qquad (\ell \neq \ell')$$
$$\rho(\partial\tau)(\ell) = \rho(\tau) \qquad \rho(c) = c$$

Fig. 7. Type-to-kind assignment definition

By choosing this ordering, we generate a model of structural, atomic subtyping. Note well that although the symbols $\bot$ and $\top$ are used, the reader should not be misled into thinking that this is a non-structural subtyping system.

$[\![Row(\tau)_A]\!]$ and $[\![Row(c)_B]\!]$: Given a finite set of labels $A \subseteq \mathcal{L}_a$, $[\![Row(\tau)_A]\!]$ is the set of total, almost constant functions from $\mathcal{L}_a\backslash_A$ into $[\![Type]\!]$. (A function is *almost constant* if it is constant except on a finite number of inputs.) In short, $Row(\tau)_A$ is the kind of rows which do *not* carry the fields mentioned in A; $Row(\tau)_\varnothing$ is the kind of complete rows. Similarly, $[\![Row(c)_B]\!]$ is the set of total, almost constant functions from $\mathcal{L}_b\backslash_B$ into $[\![Con]\!]$, so that $Row(c)_B$ is the kind of set types which do *not* carry the elements mentioned in B, and $Row(c)_\varnothing$ is the kind of complete set types. The ordering $\leq$ is extended inductively to $[\![Row(c)_B]\!]$ and $[\![Row(\tau)_A]\!]$, mutually with $[\![Type]\!]$, pointwise and covariantly as follows:

$$\frac{\hat\tau, \hat\tau' \in [\![Row(\tau)_A]\!] \qquad \forall a \in \mathcal{L}_a\backslash_A\,.\,\hat\tau(a) \leq \hat\tau'(a)}{\hat\tau \leq \hat\tau'}$$

$$\frac{\hat\tau, \hat\tau' \in [\![Row(c)_B]\!] \qquad \forall b \in \mathcal{L}_b\backslash_B\,.\,\hat\tau(b) \leq \hat\tau'(b)}{\hat\tau \leq \hat\tau'}$$

$[\![Type]\!]$: The elements of $[\![Type]\!]$ are contained in the free algebra generated by the constructors $\to$, with signature $[\![Type]\!] \times [\![Type]\!] \to [\![Type]\!]$, and $\{\cdot\}$ and $\{\cdots\}$, with signatures $[\![Row(\tau)_\varnothing]\!] \to [\![Type]\!]$ and $[\![Row(c)_\varnothing]\!] \to [\![Type]\!]$, respectively. The ordering $\leq$ is inductively extended, mutually with $[\![Row(\tau)_A]\!]$, to $[\![Type]\!]$ by treating the constructor $\to$ as contravariant in the first argument and covariant in the second, and by treating the constructors $\{\cdot\}$ and $\{\cdots\}$ as covariant; that is:

$$\frac{\hat\tau_1' \leq \hat\tau_1 \qquad \hat\tau_2 \leq \hat\tau_2'}{\hat\tau_1 \to \hat\tau_2 \leq \hat\tau_1' \to \hat\tau_2'} \qquad \frac{\hat\tau \leq \hat\tau'}{\{\hat\tau\} \leq \{\hat\tau'\}} \qquad \frac{\hat\tau \leq \hat\tau'}{\{\cdot\hat\tau\cdot\} \leq \{\cdot\hat\tau'\cdot\}}$$

This completes the definition of the model.

$$\frac{}{\rho \vdash \mathbf{true}} \qquad \frac{\rho \vdash C_1 \qquad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2} \qquad \frac{\rho = \rho' [\alpha] \qquad \rho' \vdash C}{\rho \vdash \exists \alpha . C}$$

$$\frac{\rho(\tau) = \rho(\tau')}{\rho \vdash \tau = \tau'} \qquad \frac{\rho(\tau) \leq \rho(\tau')}{\rho \vdash \tau \leq \tau'} \qquad \frac{c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

$$\frac{\tau, \tau', \tau'' : Row(c)_B \qquad \forall b \in \mathcal{L}_b \backslash_B . (c \leq \rho(\tau)(b) \Rightarrow \rho(\tau')(b) \leq \rho(\tau'')(b))}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Fig. 8. Interpretation of constraints

3.3 Interpretation in the model

We may now give the interpretation of types and constraints within the model. It is parameterized by an *assignment* ρ, i.e. a function which, for every kind k, maps $\mathcal{V}_k$ into $[\![k]\!]$. The interpretation of types is obtained by extending ρ so as to map every type of kind k to an element of $[\![k]\!]$, as defined in Fig. 7. Fig. 8 defines the constraint satisfaction predicate $\cdot \vdash \cdot$, whose arguments are an assignment ρ and a constraint C. (The notation $\rho = \rho' [\alpha]$ means that ρ and ρ' coincide except possibly on α.) These rules are not particularly surprising, except those that involve conditional constraints of the form if $c \leq \tau$ then $\tau' \leq \tau''$, where τ, τ' and τ'' are row types; we call these *complex* conditional constraints. These allow a finite specification of a countably infinite collection of conditional constraints, decomposed pointwise from the initial complex conditional, with c being the lower bound in the condition of each constraint in the collection. We clarify the meaning and utility of complex conditional constraints in Example 4.1. Constraint *entailment* is defined as usual: $C \Vdash C'$ (read: C entails C') holds iff, for every assignment ρ, $\rho \vdash C$ implies $\rho \vdash C'$. We write $C \Vdash \forall\bar{\alpha}[D].\tau$ (read: $\forall\bar{\alpha}[D].\tau$ is consistent with respect to C) iff $C \Vdash \exists\bar{\alpha}.D$.

We refer to the type and constraint logic, together with its interpretation, as RS. More precisely, we have defined two logics, where $\leq$ is interpreted as either equality or as a non-trivial subtype ordering. We will refer to them as $\mathrm{RS}^=$ and $\mathrm{RS}^\leq$, respectively. Both are sound term constraint systems [7].

3.4 Abbreviated set types

Although the set types defined in previous sections are expressive, and the form of their contents as kinds of row types allows re-use of existing implementations, an abbreviation of their form is possible. We now define a more readable, succinct form of set types as syntactic sugar for primitive set types. Each field $b : \tau$ is shortened to $b\tau$. We also define abbreviated row type constructors $\varnothing$ and ω, specifying that all elements not otherwise mentioned in a row are absent or present, respectively. For example, the set $\{r_1, r_2\}$ will be one (and the only) value of type $\{r_1+, r_2+, \varnothing\}$. Formally, the grammar for abbreviated set types is defined as follows:

$$\varsigma \quad ::= \quad \{\varsigma\} \mid b\tau, \varsigma \mid \omega \mid \varnothing \mid \beta \qquad \textit{abbreviated set types}$$

The interpretation of abbreviated set types $(\!|\, \varsigma \,|\!)$ as primitive set types is defined as follows:

$$\begin{aligned}
(\!|\, \{\varsigma\} \,|\!) &= \{\cdot (\!|\, \varsigma \,|\!) \cdot\} \\
(\!|\, b\tau, \varsigma \,|\!) &= (b : \tau \,;\, (\!|\, \varsigma \,|\!)) \\
(\!|\, \varnothing \,|\!) &= \partial - \\
(\!|\, \omega \,|\!) &= \partial + \\
(\!|\, \beta \,|\!) &= \beta
\end{aligned}$$

We say that an abbreviated set type ς is well-kinded iff $(\!|\, \varsigma \,|\!)$ is, and we write $\rho(\varsigma)$ to denote $\rho((\!|\, \varsigma \,|\!))$. In the presentation of the type system for pml_B, we will use abbreviated set types for a more succinct and readable presentation; however, we note that their definition as syntactic sugar for primitive set types allows for an implementation that re-uses row type implementations.

4 Types for pml_B

To define a type system for pml_B, we instantiate $\mathrm{HM}(X)$ with one of RS^{rel}, where *rel* ranges over $\{=, \leq\}$, and postulate records, sets, and associated operations, along with their semantics, as extensions of the core $\mathrm{HM}(X)$ language. We also define initial type bindings for these extensions, which we prove sound. This obtains a sound type system for pml_B— more than one, in fact, since our choice of *rel* results in either a unification or subtyping-based system.

4.1 Constants and initial type bindings for pml_B

To begin our conception of pml_B as an extension of the core $\mathrm{HM}(X)$ language, we postulate the constant $\{\cdot\}$, and the families of constants $\cdot.a$ and $\cdot\{a = \cdot\}$, with semantics as defined in Fig. 2. Thus, we define the constants of pml_B, along with their initial type bindings, in Fig. 9.

As is evident in Fig. 9, we make extensive use of complex conditional constraints to provide accurate types for set operations. To demonstrate how these work, and their usefulness in this context, we give the following example.

Example 4.1 Let the sets B_1 and B_2 be defined as follows:

$$\begin{aligned}
B_1 &= \{b_1, b_2, b_3\} \\
B_2 &= \{b_1, b_2, b_4\}
\end{aligned}$$

$$
\begin{aligned}
\{\cdot\} &: \forall\alpha.\alpha \to \{\partial\alpha\} \\
\cdot\{a = \cdot\} &: \forall\alpha_1\alpha_2\beta.\{a : \alpha_1\,;\ \beta\} \to \alpha_2 \to \{a : \alpha_2\,;\ \beta\} \\
\cdot.a &: \forall\alpha\beta.\{a : \alpha\,;\ \beta\} \to \alpha \\
B &: \{B+, \varnothing\} \\
\bar{B} &: \{B-, \omega\} \\
\ni b &: \forall\beta.\{b+, \beta\} \to \{b+, \beta\} \\
\wedge &: \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
&\qquad \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \varnothing \leq \beta_3 \\
&\qquad\qquad\qquad\quad \wedge\ \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \\
\vee &: \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
&\qquad \text{where } C = \quad \text{if } + \leq \beta_1 \text{ then } \omega \leq \beta_3 \\
&\qquad\qquad\qquad\quad \wedge\ \text{if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \\
\ominus &: \forall\beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\
&\qquad \text{where } C = \quad \text{if } + \leq \beta_2 \text{ then } \varnothing \leq \beta_3 \\
&\qquad\qquad\qquad\quad \wedge\ \text{if } - \leq \beta_2 \text{ then } \beta_1 \leq \beta_3 \\
?_b &: \forall\bar{\alpha}\bar{\beta}\gamma[C].\{b\gamma, \beta\} \to (\{b+, \beta_1\} \to \alpha_1) \to (\{b-, \beta_2\} \to \alpha_2) \to \alpha \\
&\qquad \text{where } C = \quad \text{if } + \leq \gamma \text{ then } \beta \leq \beta_1 \wedge \text{ if } - \leq \gamma \text{ then } \beta \leq \beta_2 \\
&\qquad\qquad\qquad\quad \wedge\ \text{if } + \leq \gamma \text{ then } \alpha_1 \leq \alpha \wedge \text{ if } - \leq \gamma \text{ then } \alpha_2 \leq \alpha
\end{aligned}
$$

Fig. 9. Constants and initial type bindings for pml_B

Suppose then that we wish to type the expression $B_1 \wedge B_2$, using the unification-based constraint system $\mathrm{RS}^=$. Given the typing for $\wedge$ defined in Fig. 9, the variables β_1 and β_2 will be unified with the types of the contents of B_1 and B_2, respectively:

$$\beta_1 = (b_1+, b_2+, b_3+, \varnothing)$$

$$\beta_2 = (b_1+, b_2+, b_4+, \varnothing)$$

Additionally, β_3 will be unified with a type that is "splittable" into the appropriate form for the expansion of the complex conditional constraint in the type of $\wedge$:

$$\beta_3 = (b_1\gamma_1, b_2\gamma_2, b_3\gamma_3, b_4\gamma_4, \beta)$$

Then, given the rules for complex conditional constraints defined in Fig. 8, which specify that any complex conditional constraint is decomposable pointwise into an arbitrary length conjunction of "normal" conditional constraints, the constraint C in the type of $\wedge$ can be expanded as follows:

$$
\begin{aligned}
C = \quad & \text{if } - \leq + \text{ then } - \leq \gamma_1 \wedge \text{ if } + \leq + \text{ then } + \leq \gamma_1 \\
\wedge\ & \text{if } - \leq + \text{ then } - \leq \gamma_2 \wedge \text{ if } + \leq + \text{ then } + \leq \gamma_2 \\
\wedge\ & \text{if } - \leq + \text{ then } - \leq \gamma_3 \wedge \text{ if } + \leq + \text{ then } - \leq \gamma_3 \\
\wedge\ & \text{if } - \leq - \text{ then } - \leq \gamma_4 \wedge \text{ if } + \leq - \text{ then } + \leq \gamma_4 \\
\wedge\ & \text{if } - \leq \varnothing \text{ then } \varnothing \leq \beta \wedge \text{ if } + \leq \varnothing \text{ then } \varnothing \leq \beta
\end{aligned}
$$

This expansion will force the following unification:

$$
\begin{aligned}
\beta_3 &= (b_1+, b_2+, b_3-, b_4-, \varnothing) \\
& \quad \textit{or} \\
\beta_3 &= (b_1+, b_2+, \varnothing)
\end{aligned}
$$

And this in fact is the type of $\{b_1, b_2\}$, and $B_1 \wedge B_2 \to \{b_1, b_2\}$.

4.2 Type soundness for pml_B

Given the previous development, we may now define the type systems for pml_B. Specifically, these are denoted $\mathcal{S}^=$ and $\mathcal{S}^\leq$ and obtained from extending HM(RS$^=$) and HM(RS$^\leq$), respectively, with the constants and type bindings defined in Fig. 9 and the associated semantics, defined in Fig. 2. To prove syntactic type soundness for HM(X) in either of $\mathcal{S}^{rel}$, we demonstrate the requisite δ-typability lemma, which means that the initial type bindings for the pml_B constants are sound:

Lemma 4.2 pml_B *is δ-typable in $\mathcal{S}^{rel}$.*

The proof is straightforward, and is omitted here for brevity; interested readers are referred to [15] for details. Given this lemma, the soundness of RS^{rel}, and results demonstrated in [16], we immediately obtain type soundness for pml_B in both type systems:

Theorem 4.3 (pml_B Type Soundness) *If e is a pml_B expression which is well-typed in $\mathcal{S}^{rel}$, then e does not go wrong.*

In addition to this result, the systems $\mathcal{S}^{rel}$ enjoy the benefits of type inference: HM(X) provides a type inference algorithm modulo constraint solution [7], and a row type and conditional constraint solution algorithm exists and has been proven correct [8].

A consequence of Theorem 4.3 is that certain pml_B runtime optimizations may be effected. For example, this result implies that all membership checks

$\ni_b$ may be removed at runtime from a well-typed program. This property is verified by the following result, which follows by type soundness:

Proposition 4.4 *Let $\rightsquigarrow$ be defined as $\rightarrow$, but with the memcheck rule redefined as $B \ni b \rightsquigarrow B$; that is, no runtime membership checks are performed. Suppose e is well typed; then $e \rightsquigarrow^\star v$ iff $e \rightarrow^\star v$.*

5 Characterizing pml_B: enumeration encoding

In this section we show that sets are a dual of enumerations (nullary variant constructors); intuitively, sets are a conjunction of elements, whereas enumerations are a disjunction of elements. This is another obvious instance of the well-known duality between and-or, records-variants, and products-coproducts. The asymmetric nature of programs and continuations causes the duality between records and variants [1,6] to be imperfect: while the type-indexed rows of [14] are able to encode both records and variants, they are more expressive than both records and variants, and the fact remains that neither can fully and faithfully be encoded using the other. However, the situation is better with set and enumeration types. There is still an asymmetry, in that records are needed to encode enumerations, but enumeration types can fully and faithfully encode set types. The main point of this characterization is that it illuminates the expressiveness of pml_B, reinforcing our argument that pml_B types can be used to give expressive static types for languages with primitive program labels.

To begin, we imagine a new functional language core containing syntax for expressing enumerations (where $`b$ is an injection of b into an enumeration):

$$e ::= `b \mid \text{match } e \text{ with } b \rightarrow e \mid (\text{match } e \text{ with } b \rightarrow e \mid \_ \rightarrow e) \mid \cdots$$

Note that in match expressions with defaults, the vertical bar is part of the language syntax, not the grammar. Long match expressions may be expressed by "chaining" shorter matches; for ease of presentation we define the following syntactic sugar:

$$\text{match } e \text{ with } b_1 \rightarrow e_1 \mid b_2 \rightarrow e_2 \mid \cdots \mid b_n \rightarrow e_n \mid \_ \rightarrow e_0$$

$$\triangleq$$

$$\text{match } e \text{ with } b_1 \rightarrow e_1 \mid \_ \rightarrow \text{match } e \text{ with } b_2 \rightarrow e_2 \mid \_ \rightarrow \cdots$$
$$\text{match } e \text{ with } b_n \rightarrow e_n \mid \_ \rightarrow e_0$$

5.1 Encoding enumerations with records

We will give the term-level encodings only, but the translations are faithful with respect to typing as well. The encoding of enumerations as records requires us to freeze the encoded results of matches using abstractions, to thaw the appropriate result in the case of a match, and also to ensure failure in the case of a mismatch:

$$\textbf{freeze } e \triangleq \lambda x.e \qquad \text{where } x \text{ not free in } e$$

$$\textbf{thaw } e \triangleq e\,\{\} \qquad \textbf{fail} \triangleq \varnothing \ni b$$

$$\begin{aligned}
[\![`b]\!] &= \lambda r.\,(\textbf{thaw } r.a_b)\\
[\![\text{match } e_1 \text{ with } b \to e_2]\!] &= [\![e_1]\!](\{\textbf{freeze fail}\}\{a_b = \textbf{freeze } [\![e_2]\!]\})\\
[\![\text{match } e_1 \text{ with } b \to e_2 \mid _ \to e_3]\!] &= [\![e_1]\!](\{\textbf{freeze } [\![e_3]\!]\}\{a_b = \textbf{freeze } [\![e_2]\!]\})\\
&\vdots
\end{aligned}$$

And so on trivially for the other functional language constructs. Since pml_B comes with an accurate static analysis of records, this encoding suggests an alternate, typed implementation of enumerations, as well as any higher-level language with a labeling system that can be expressed via enumerations.

5.2 Encoding sets with enumerations

Since sets in pml_B come with a rich library of operations, the transformation from sets to enumerations is a bit trickier. However, it can be done in such a way that set membership checks via $\ni_b$ succeed or fail consistently in the encoding, and likewise set membership tests via $?_b$ branch consistently. The encoding uses appropriately defined combinators to implement set operations, where fresh elements are returned by calls to $\textbf{fresh}_b()$ during the encoding. First, we specify the following macros, intended to define the operational behavior of succeeding and failing set membership checks, respectively:

$$\textbf{succeed} \triangleq \lambda x.()$$

$$\textbf{fail} \triangleq \lambda x.\text{match } x \text{ with } \textbf{fresh}_b() \to ()$$

In the definition of **fail**, the label generated by the call $\textbf{fresh}_b()$ will be distinct from any other label in a given program, so that any application of **fail** will cause an operational match failure. Further, in a sound variant type system, any use of **fail** will be statically rejected, meaning that failing set membership checks are statically rejected in the encoding— thus, the encoding also preserves type safety. If a purely operational encoding of pml_B is desired, we can alternately define **fail** in a language with exceptions as $\textbf{fail} \triangleq \lambda x.\text{raise Fail}$.

The rest of the encoding is given as follows; note the use of **succeed** and

fail in any membership check:

$$
\begin{aligned}
[\![\{b_1,\ldots,b_n\}]\!] &= \lambda f.\lambda g.\lambda x.\,\text{match } x \text{ with } b_1 \to f(x) \mid \\
&\qquad\qquad \vdots \\
&\qquad\quad b_n \to f(x) \mid \_ \to g(x) \\
[\![\{\overline{b_1,\ldots,b_n}\}]\!] &= \lambda f.\lambda g.\lambda x.\ \text{match } x \text{ with } b_1 \to g(x) \mid \\
&\qquad\qquad \vdots \\
&\qquad\quad b_n \to g(x) \mid \_ \to f(x) \\
[\![\wedge]\!] &= \lambda s_1.\lambda s_2.\lambda f.\lambda g.\ s_1(s_2 f\ g)\ g \\
[\![\vee]\!] &= \lambda s_1.\lambda s_2.\lambda f.\lambda g.\ s_1 f\ (s_2 f\ g) \\
[\![\ominus]\!] &= \lambda s_1.\lambda s_2.\lambda f.\lambda g.\ s_1(s_2\, g\, f)\ g \\
[\![\ni_b]\!] &= \lambda s.\ s\ \textbf{succeed}\ \textbf{fail}\ \text{`}b \\
[\![?_b]\!] &= \lambda s.\lambda y.\lambda z.\ s\,(\lambda x.y)(\lambda x.z)\,\text{`}b \\
&\vdots
\end{aligned}
$$

And so on trivially for the other functional language constructs. At an operational level, this encoding suggests an alternate implementation of sets in e.g. OCaml, which comprises a rich language of variants. However, while polymorphic variant types [3,11] do exist in OCaml, they are not expressive enough to accurately type the above encoding, and so provide an alternate implementation of our set type system. For this, it would be sufficient to extend the OCaml variant type system with conditional constraints, an addition that has been described in [4]. If the reader is interested in exploring this encoding at the operational level in OCaml, the alternate definition of **fail** given above should be used.

6 Applications

In this section we discuss several applications of pml_B as an implementation language for higher-level languages, demonstrating its usefulness as a uniform foundation for this purpose. In particular, the implementation of these languages in pml_B provides them with a sound, indirect type analysis with no additional theoretical overhead, via composition of their translation into pml_B and the pml_B type system. Furthermore, the precise static analysis of pml_B facilitates run-time optimizations in the source languages. The pml_B type system can also serve as a theoretical basis for the development of direct source language type systems, saving significant effort in the proof of direct type safety.

6.1 Stack inspection

In [10] the authors implement the so-called security-passing-style transformation from a language with stack inspection security, called λ_{sec}, into a less general form of pml_B, called λ_{set}. That presentation uses binary set operations for which one element is always statically known, and does not require complex conditional constraints for typings; the current presentation is thus an extension in this regard. The λ_{sec}-to-λ_{set} transformation provides an indirect static analysis for λ_{sec}, meaning that runtime security checks can be eliminated, improving efficiency of the language. This transformation also allows certain compiler optimizations, such as CPS and tail-call optimizations, which are otherwise prevented by the requirements of the runtime stack inspection algorithm. In addition, the transformation drives the development of a direct static analysis for λ_{sec}, providing insight into its form and greatly easing proof of its soundness. By proving that the transformation preserves semantics, it is sufficient to demonstrate a simple syntactic correspondence between direct λ_{sec} type judgements and pml_B type judgements of transformed λ_{sec} terms; the much more complicated route of proving subject reduction for the direct λ_{sec} type system is unnecessary.

In the stack inspection security model, code owners p are associated with regions of code $p.e$. Each code owner is locally associated with sets of privileges R via an access-control-list $\mathcal{A}$. Individual privileges r are explicitly activated via expressions of the form $\mathsf{enable}\, r \,\mathsf{in}\, e$, and their activation may be checked via expressions of the form $\mathsf{check}\, r \,\mathsf{then}\, e$. Security information is maintained on the call-stack, and privilege checks are implemented via the stack inspection algorithm, which analyzes the call-stack.

For example, on a local system, identified as p_s, we might wish to make printing a privileged resource. To enforce this, we could provide a safeprint function, which interposes a check of the **PrintPriv** privilege before printing:

$$\text{safeprint} \;=\; \lambda x.p_s.\mathsf{check}\,\mathbf{PrintPriv}\,\mathsf{then}\,\text{print(x)}$$

Printing could then be accomplished in an environment with **PrintPriv** enabled, by using safeprint in code owned by principals locally authorized for **PrintPriv**. The stack inspection algorithm prevents any code owned by principals *not* authorized for **PrintPriv** from gaining unauthorized access to printing through man-in-the-middle attacks.

In the security-passing-style, security information is essentially passed down the stack, rather than maintained on it. In our transformation, denoted $[\![e]\!]_p$, with p the owner of e, security information is maintained in the program variable s. Privilege activation is encoded as an addition to s:

$$[\![\mathsf{enable}\, r \,\mathsf{in}\, e]\!]_p \;=\; \mathsf{let}\, s = s \vee (\{r\} \cap \mathcal{A}(p))\, \mathsf{in}\, [\![e]\!]_p$$

Note that this addition only occurs if the code owner is authorized for the privilege. Privilege checking then becomes a matter of simply checking for the presence of the privilege in s:

$$[\![\mathsf{check}\, r\, \mathsf{then}\, e]\!]_p \;=\; \mathsf{let}\, \_ = s \ni r\, \mathsf{in}\, [\![e]\!]_p$$

When a new code owner p' is encountered, s is intersected with the privileges granted to p' locally, to fully enforce the stack-inspection model:

$$[\![p'.e]\!]_p \;= \mathsf{let}\, s = s \wedge \mathcal{A}(p')\, \mathsf{in}\, [\![e]\!]_{p'}$$

A significant feature of this translation is that the λ_{set} type system— subsumed in the pml_B system— provides a precise specification of sets, so that all privilege checks are statically enforced. This means that program execution can be made more efficient, since dynamic privilege checks are not necessary.

6.2 Object confinement

In [17] the authors define a language model for expressing various object confinement mechanisms, called pop. An indirect static analysis is obtained for pop by transformation into pml_B, utilizing most of the language and type features presented here. However, pml_B and its type system is not formally defined there, nor proven sound. This presentation provides these results, with the caveat that state is not treated— but this is a minor detail, since the presentation of HM(X) in [16] contains state as part of the core calculus. The correctness of our general approach to encoding an OO language in a language with records, state, and a constraint-based type system is established in [2]. As is the case for λ_{sec}, the pop-to-pml_B transformation also drives the development of a direct type system for pop, and eases its soundness proof, while the precise pml_B type discipline allows optimizations via the static enforcement of dynamic security checks.

The pop system is an object-based calculus, where each object is assigned a domain label d. These may be interpreted in various ways— as e.g. code owners, or regions of static scope— allowing the language to model a variety of approaches to security. Objects are also endowed with a user interface φ, which is a mapping from domain labels to sets of method names, and specifies the per-domain access rights for objects; default access rights are specified via the "wildcard" domain ∂, and are universally authorized. For example, a file object o may be defined as follows, which is read/write in its own domain, but read-only otherwise:

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

Security is then enforced on a *use* basis: when an object is used in a particular domain, runtime checks ensure that the use is authorized. Continuing the above example, the use $o.\text{read}()$ is allowed in domain $d' \neq d$, but $o.\text{write}(e)$ is not. This use-based approach has various benefits, e.g. a more fine-grained specification of access rights.

Highlights of the pop-to-pml$_B$ transformation are then defined as follows. The transformation of interfaces φ is denoted $\hat{\varphi}$, and uses records of sets in the image:

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

Objects are transformed roughly as follows[3], where advanced features such as the treatment of "self" are omitted here for brevity:

$$\begin{gathered} [\![[m_1(x) = e_1, \ldots, m_n(x) = e_n] \cdot d \cdot \varphi]\!]_{d'} \\ = \\ \{\mathrm{obj} = \{m_1 = \lambda x.[\![e_1]\!]_d, \ldots, m_n = \lambda x.[\![e_n]\!]_d\},\ \mathrm{ifc} = \hat{\varphi}\} \end{gathered}$$

Then, method selects are encoded so that access rights are always verified; set union is used to guarantee default access to the object:

$$\begin{aligned} [\![e_1.m(e_2)]\!]_d \;=\; & \mathsf{let}\, c_1 = [\![e_1]\!]_d \,\mathsf{in} \\ & (c_1.\mathrm{ifc}.d \vee c_1.\mathrm{ifc}.\partial) \ni m; \\ & (c_1.\mathrm{obj}.m)([\![e_2]\!]_d) \end{aligned}$$

Again, a significant feature of this transformation is that it highlights the applicability of the pml$_B$ type analysis to security; all access checks associated with method invocation are statically verified, and may be removed at run-time. This applicability extends to other language features, including access restriction mechanisms implemented with intersection.

7 Conclusion

In this paper we have defined the pml$_B$ programming language, whose principal novelty is a set of features for defining finite and cofinite sets of urelements and associated operations, along with a type system that exploits row types and conditional constraints to accurately type these features. The type system is defined by instantiation of HM(X), which provides an easy method for proof of type soundness and definition of type inference.

We also describe applications of pml$_B$ as an implementation language for other systems, including a language incorporation stack inspection, and a

[3] In these transformations, we use the following syntactic sugar:

$$\begin{aligned} \{m_1 = e_1, \ldots, m_n = e_n\} &\triangleq \{\varnothing\}\{m_1 = e_1\} \cdots \{m_n = e_n\} \\ e_1; e_2 &\triangleq \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \qquad x \text{ not free in } e_2 \end{aligned}$$

language for modeling object confinement mechanisms. Via these implementations, the precise pml_B type discipline provides a static analysis of dynamic properties for these systems, allowing runtime optimizations. These distinct applications also suggest that there may be other uses of pml_B as an implementation language for higher-level systems, indeed any system that contains labels and associated operations as a central part of its feature set.

References

[1] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.URL: `http://research.microsoft.com/Users/luca/Papers/Inheritance.ps`.

[2] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[3] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998. URL: `wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/variants.ps.gz`.

[4] Jacques Garrigue. Simple type inference for structural polymorphism. In *Proceedings of the 9th Workshop on Foundations of Object-Oriented Languages (FOOL9)*, Portland, OR, January 2002. Revised 12/11/2002. URL: `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/structural-inf-us.ps.gz`.

[5] Nevin Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, pages 189–206, 1995. URL: `citeseer.nj.nec.com/article/heintze95controlflow.html`.

[6] Susumu Nishimura. Static typing for dynamic messages. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 266–278, New York, NY, 1998. URL: `citeseer.nj.nec.com/nishimura98static.html`.

[7] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps`.

[8] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000. URL: `http://pauillac.inria.fr/~fpottier/wallace/`.

[9] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz`.

[10] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-skalka-smith-esop01.ps.gz`.

[11] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[12] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`.

[13] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.

[14] Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–275. ACM Press, 2001. URL: `http://doi.acm.org/10.1145/360204.360230`.

[15] Christian Skalka. *Types for Programming Language-Based Security*. PhD thesis, The Johns Hopkins University, August 2002. URL: `http://www.cs.uvm.edu/~skalka/skalka-pubs/skalka-phd-thesis.ps`.

[16] Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Proceedings of the 2002 Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, Dagstuhl, Germany, July 2002. URL: `http://pauillac.inria.fr/~fpottier/publis/skalka-fpottier-tip-02.ps.gz`.

[17] Christian Skalka and Scott Smith. Static use-based object confinement. In *Workshop on Foundations of Computer Security (FCS02)*, 2002. URL: `http://www.cs.jhu.edu/~ces/papers/skalka-smith-fcs02.ps`.

[18] Scott Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Verlag, March 2000. URL: `http://link.springer.de/link/service/series/0558/papers/1782/17820382.pdf`.

[19] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 200X. To appear. URL: `http://types.bu.edu/reports/Wel+Dim+Mul+Tur:JFP-ta.html`.