Set Types and Applications

Christian Skalka ¹

The University of Vermont

Scott Smith ²

The Johns Hopkins University

Abstract

We present pml, 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 by showing how it can serve as a practical implementation language for higher-level systems: it can express several forms of programming language-based security, and implement inferred monomorphic flow types.

1 Introduction

In this paper we present the pml 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 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 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.

The pml language of records includes default values in the style of Rémy’s Projective ML [10]. 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@emba.uvm.edu
² Email: scott@cs.jhu.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
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 [13], instantiating it with a constraint system containing row types [11] and conditional constraints [2]; 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 [8][14].

The language and type system presented here is a more general version of the $\lambda_{\text{set}}$ language and type system, defined in [8]. In that presentation, one element of set union and intersection operations was always statically known; in pml, these operations are fully generalized. The pml language also extends $\lambda_{\text{set}}$ with a general set difference operation, co-finite 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 set operations, and we define an abbreviated type form for more succinct and readable set types.

The pml 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 and allowing run-time checks to be eliminated as a result of type safety. Thus, the pml language facilitates run-time optimizations of source languages. We discuss several examples of this, including an implementation of Java-style stack inspection [8], and an implementation of an Object-Oriented language model with object confinement mechanisms [14]. We also define an encoding of flow types [3][16], a system that allows flow-based optimizations in compilers. One contribution of this encoding is the benefit of an inferred type analysis.

To characterize the expressiveness of the pml 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 language: syntax and semantics

The grammar for pml is given in Fig. 1, the semantics in Fig. 2. The language is based on Rémy’s Projective ML [10], containing records with default values, manipulated with the elevation and modification record constructors \{e\} and e\{a = e’\}, and the projection destructor e.a.

We take as a 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 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 $\overline{B}$. This latter feature presents some practical implementation issues, but in this presenta-


\[
x \in \mathcal{V}, \ a \in \mathcal{L}_a, \ b \in \mathcal{L}_b, \ B \subseteq \mathcal{L}_b
\]

\[
v :: \ fix \ z. \lambda x.e \ | \ s \ | \ \{v\} \ | \ v\{a = v\}
\]

\[
s :: B \ | \ \bar{B} \ | \ \vee \ | \ \land \ | \ \ominus \ | \ \exists_b \ ?_b
\]

\[
e :: x \ | \ v \ | \ e e \ | \ let \ x = \text{in} \ e \ | \ \{e\} \ | \ e\{a = e\} \ | \ e.a
\]

\[
E :: [ ] \ | \ E e \ | \ v E \ | \ \{E\} \ | \ E\{a = e\} \ | \ v\{a = E\} \ | \ E.a
\]

---

Fig. 1. Grammar for pml

In addition we take it at mathematical "face value"—that is, we take \( \bar{B} \) to denote \( \mathcal{L}_b \setminus B \). Basic set operations are provided, including \( \exists_b, \ \land, \ \vee \) 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 \( \exists_b \). For clarity of presentation, we define the following syntactic sugar:

\[
(\exists_b \ e) \triangleq (e \exists b)
\]

\[
(\land e_1 e_2) \triangleq (e_1 \land e_2)
\]

\[
(\lor e_1 e_2) \triangleq (e_1 \lor e_2)
\]

\[
(\ominus e_1 e_2) \triangleq (e_1 \ominus e_2)
\]

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

3 The type constraint system RS

We define the type system for pml as an instance of the HM(\( X \)) framework \[13,5\]. 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. 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 \( \delta \)-typability 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 is defined, in part, by instantiating HM(\( X \)) with the RS type constraint system, comprising row types and conditional
\[(\text{fix } z. \lambda x. e)v \rightarrow e[v/x][\text{fix } z. \lambda x. e/z] \quad (\beta)\]

\[\text{let } x = v \text{ in } e \rightarrow e[v/x] \quad (\text{let})\]

\[\{v\}.a \rightarrow v \quad (\text{default})\]

\[v_1\{a = v_2\}.a \rightarrow v_2 \quad (\text{access})\]

\[v_1\{a' = v_2\}.a \rightarrow v_1.a \quad a' \neq a \quad (\text{skip})\]

\[B \ni b \rightarrow B \quad \text{if } b \in B \quad (\text{memcheck})\]

\[B_1 \land B_2 \rightarrow B_1 \cap B_2 \quad (\text{intersect})\]

\[B_1 \lor B_2 \rightarrow B_1 \cup B_2 \quad (\text{union})\]

\[B_1 \ominus B_2 \rightarrow B_1 - B_2 \quad (\text{difference})\]

\[?_b B \rightarrow \lambda f. \lambda g. f(B) \quad \text{if } b \in B \quad (\text{memtesty})\]

\[?_b B \rightarrow \lambda f. \lambda g. g(B) \quad \text{if } b \not\in B \quad (\text{memtestn})\]

\[E[e] \rightarrow E[e'] \quad \text{if } e \rightarrow e' \quad (\text{context})\]

Fig. 2. Operational semantics for pml

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 \(\ell\) ranges over \(L_a \cup L_b\). 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 rows. Row types are built up using the usual constructors, including \(\partial_{\tau}\) which specifies that all fields not otherwise mentioned in a row have type \(\tau\). The original presentation of rows [10][11] includes an equational theory, which in particular allow rows to commute. 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 \(\rho\) using the record type constructor \(\{\rho\}\). Set types are also built up from a particular form of row types, using the set type constructor \(\{\cdot \rho\}\). 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 (⊥) (NB: ⊥ and \(\top\) here are not the same as the “top” and “bottom” types.
\[
\begin{array}{l}
\text{VAR} \quad \Gamma(x) = \sigma \quad C \vdash \sigma \\
\hline
C, \Gamma \vdash x : \sigma
\end{array}
\quad \begin{array}{l}
\text{SUB} \quad C, \Gamma \vdash e : \tau \\
\hline
C, \Gamma \vdash \tau \leq \tau'
\end{array}
\]
\[
\begin{array}{l}
\text{ABS} \quad C, (\Gamma; x : \tau) \vdash e : \tau' \\
\hline
C, \Gamma \vdash \lambda x.e : \tau \rightarrow \tau'
\end{array}
\quad \begin{array}{l}
\text{APP} \quad C, \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad C, \Gamma \vdash e_2 : \tau_2 \\
\hline
C, \Gamma \vdash e_1 e_2 : \tau
\end{array}
\]
\[
\begin{array}{l}
\text{LET} \quad C, \Gamma \vdash e_1 : \sigma \\
\hline
C, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\end{array}
\quad \begin{array}{l}
\text{\forall INTRO} \quad C \land D, \Gamma \vdash e : \tau \\
\hline
C \land \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D], \tau
\end{array}
\quad \begin{array}{l}
\text{\forall ELIM} \quad C, \Gamma \vdash e : \forall \bar{\alpha}[D], \tau' \quad C \vdash [\bar{\tau}/\bar{\alpha}]D \\
\hline
C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'
\end{array}
\]

Fig. 3. The system HM(\(X\))

\[
\tau ::= \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \{\tau\} \mid \ell : \tau \mid \partial \tau \mid \tau \ref \mid c
\]

\text{types}
\[
c ::= + \mid - \mid \top \mid \bot
\]

\text{constructors}
\[
C ::= \text{true} \mid C \land C \mid \exists \alpha.C \mid \tau = \tau \mid \tau \leq \tau \mid \text{if } c \leq \tau \text{ then } \tau \leq \tau
\]

\text{constraints}

Fig. 4. RS Grammar

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 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 being built up from a specific kind of rows, is that set types can be soundly implemented by \textit{re-use} of existing row type implementations.

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

\[
k ::= \text{Con} \mid \text{Row}(\tau)_A \mid \text{Row}(c)_B \mid \text{Type}
\]

where \(A\) ranges over finite subsets of field labels \(\mathcal{L}_a\) and \(B\) ranges over finite
Fig. 5. Kinding rules for RS types

\[
\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \quad \frac{\tau : Type}{\tau \text{ ref} : Type} \quad \frac{\tau, \tau' : Type}{\tau \to \tau' : Type} \quad \frac{\tau : \text{Row}(\tau)_{\emptyset}}{\{\cdot \tau\} : Type} \quad \frac{\tau : Type}{\partial \tau : \text{Row}(\tau)_A} \quad \frac{\tau : \text{Type}}{c : \text{Con}}
\]

\[
\frac{\tau : \text{Con}}{b \notin B} \quad \frac{\tau' : \text{Row}(c)_{B \cup \{b\}}}{(b : \tau ; \tau') : \text{Row}(c)_B}
\]

\[
\frac{\tau : \text{Type}}{a \notin A} \quad \frac{\tau' : \text{Row}(\tau)_{A \cup \{a\}}}{(a : \tau ; \tau') : \text{Row}(\tau)_A}
\]

Fig. 6. Kinding rules for RS constraints

\[
\frac{\vdash \text{true}}{\vdash C_1, C_2} \quad \frac{\vdash C_1 \land C_2}{\vdash C} \quad \frac{\exists \alpha. C}{\vdash C} \quad \frac{\tau, \tau' : k}{\vdash \tau = \tau'} \quad \frac{\vdash \tau \leq \tau'}{\vdash \tau \leq \tau'}
\]

\[
\frac{\tau : \text{Con}}{\vdash \text{if } c \leq \tau \text{ then } C} \quad \frac{\tau, \tau', \tau'' : \text{Row}(c)_B}{\vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}
\]

subsets of set urelements \( \mathcal{L}_b \). Row kinds are parametrized by \( \tau \) 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 purpose of these rules is to guarantee that every constraint has a well-defined interpretation within our model, defined in Sect. 3.2.

### 3.2 The model

The model for RS is constructed by associating with every kind \( k \) a mathematical structure denoted \( [k] \). Each of these structures contain elements which can be informally described as ground types—that is, variable-free types—of the relevant kind. We denote these elements \( \mathring{\tau} \). Each structure \( [k] \) is equipped with a partial ordering \( \leq \) of its elements. Accordingly, the relation on each \( [k] \) is defined to be transitive and reflexive, by requiring the following inferences
\[
\begin{array}{ll}
\rho(\tau_{\text{ref}}) = \rho(\tau)_{\text{ref}} & \rho(\tau \rightarrow \tau') = \rho(\tau) \rightarrow \rho(\tau') \\
\rho(\{\tau\}) = \{\rho(\tau)\} & \rho(\{\tau\}.) = \{\rho(\tau)\} \\
\rho(\ell : \tau ; \tau')(\ell) = \rho(\tau) & \rho(\ell : \tau ; \tau')(\ell') = \rho(\tau')(\ell') \\
\rho(\partial\tau)(\ell) = \rho(\tau) & \rho(c) = c \\
\end{array}
\]

Fig. 7. Type-to-kind assignment definition

To be axiomatic for all \(\hat{\tau}\):

\[
\hat{\tau} \leq \hat{\tau}' \qquad \hat{\tau} \leq \hat{\tau}' \leq \hat{\tau}'' \quad \frac{\hat{\tau} \leq \hat{\tau}'}{\hat{\tau} \leq \hat{\tau}''}
\]

The model is explicated for each \([k]\) as follows:

\textbf{[Con]}: The elements of \([\text{Con}]\) are contained in the set \{+, −, ⊥, ⊤\}. As is made clear by the full definition of our model, continued below, the characteristics of the ordering \(≤\) over the model is determined by the definition of \(≤\) over \([\text{Con}]\); if we define \(≤\) over \([\text{Con}]\) as equality, then \(≤\) is an equivalence relation over the entire model— that is, over each \([k]\). On the other hand, we may choose a subtype ordering over \([\text{Con}]\), axiomatized as follows:

\[
\bot \leq + \qquad \bot \leq - \qquad + \leq \top \qquad - \leq \top
\]

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.

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

\[
\hat{\tau}, \hat{\tau}' \in [\text{Row(\tau)_A}] \quad \forall a \in \mathcal{L}_a \setminus A. \hat{\tau}(a) \leq \hat{\tau}'(a) \\
\hat{\tau} \leq \hat{\tau}'
\]

\[
\hat{\tau}, \hat{\tau}' \in [\text{Row(c)_B}] \quad \forall b \in \mathcal{L}_b \setminus B. \hat{\tau}(b) \leq \hat{\tau}'(b) \\
\hat{\tau} \leq \hat{\tau}'
\]

\textbf{[Type]}: The elements of \([\text{Type}]\) are contained in the free algebra gener-
\[
\begin{array}{cccc}
\rho \vdash \text{true} & \rho \vdash C_1 & \rho \vdash C_2 & \rho = \rho' [\alpha] \\
\end{array}
\]
\[
\rho(\tau) = \rho(\tau') & \rho(\tau) \leq \rho(\tau') & c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau'' & \rho \vdash \exists \alpha. C
\]
\[
\rho \vdash \tau = \tau' & \rho \vdash \tau \leq \tau' & \rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''
\]
\[
\tau, \tau', \tau'' : \text{Row}(c)_B & \forall b \in \mathcal{L}_B \cdot (c \leq \rho(\tau)(b) \Rightarrow \rho(\tau')(b) \leq \rho(\tau'')(b))
\]
\[
\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''
\]

Fig. 8. Interpretation of constraints

ated by the constructors \(\rightarrow\), with signature \([[Type]] \times [[Type]] \rightarrow [[Type]]\), and \(\{\cdot\}\), with signature \([[\text{Row}(\tau)]] \rightarrow [[Type]]\). The ordering \(\leq\) is coinductively extended with \([[\text{Row}(\tau)]]\), to \([[Type]]\) by treating the constructor \(\rightarrow\) as contravariant in the first argument and covariant in the second, and by treating the constructors \(\{\cdot\}\) and \(\{\cdot \cdots \}\) as covariant; that is:

\[
\begin{align*}
\hat{\tau}_1' \leq \hat{\tau}_1 & \quad \hat{\tau}_2' \leq \hat{\tau}_2' \\
\hat{\tau}_1 \rightarrow \hat{\tau}_2 \leq \hat{\tau}_1' \rightarrow \hat{\tau}_2' & \quad \{\hat{\tau}\} \leq \{\hat{\tau}'\} \\
\hat{\tau} \leq \hat{\tau}' & \quad \{\cdot \hat{\tau}\} \leq \{\cdot \hat{\tau}'\}
\end{align*}
\]

This completes the definition of the model.

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 \(\rho\), i.e. a function which, for every kind \(k\), maps \(V_k\) into \([[k]]\). The interpretation of types is obtained by extending \(\rho\) 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 \(\rho\) and a constraint \(C\). (The notation \(\rho = \rho' [\alpha]\) means that \(\rho\) and \(\rho'\) coincide except possibly on \(\alpha\).) These rules are not particularly surprising, except those that involve conditional constraints of the form if \(c \leq \tau\) then \(\tau' \leq \tau''\), where \(\tau\) is a row or set type; we call these complex conditional constraints. The meaning and utility of complex conditional constraints will be demonstrated in subsection 4.1. Constraint entailment is defined as usual: \(C \vDash C'\) (read: \(C\) entails \(C'\)) holds iff, for every assignment \(\rho\), \(\rho \vdash C\) implies \(\rho \vdash C'\).

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 \(\text{RS}^=\) and \(\text{RS}^\leq\), respectively. Both are sound term constraint systems [3].
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 $\omega$, 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:

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

The interpretation of abbreviated set types $\langle \varsigma \rangle$ as primitive set types is defined as follows:

$$\langle \{\varsigma\} \rangle = \{\langle\varsigma\rangle\}$$
$$\langle b\tau, \varsigma \rangle = (b : \tau ; \langle \varsigma \rangle)$$
$$\langle \varnothing \rangle = \partial^-$$
$$\langle \omega \rangle = \partial^+$$
$$\langle \beta \rangle = \beta$$

We say that an abbreviated set type $\varsigma$ is well-kinded iff $\langle \varsigma \rangle$ is, and we write $\rho(\varsigma)$ to denote $\rho(\langle \varsigma \rangle)$. In the presentation of the type system for $\text{pml}$, 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 $\text{pml}$

To define a type system for $\text{pml}$, we instantiate $\text{HM}(X)$ with one of $\text{RS}_{rel}$, where $rel$ ranges over $\{=, \leq\}$, and postulate records, sets, and associated operations, along with their semantics, as extensions of the core $\text{HM}(X)$ language. We also define initial type bindings for these extensions, which we prove sound. This obtains a sound type system for $\text{pml}$—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 $\text{pml}$

To begin our conception of $\text{pml}$ as an extension of the core $\text{HM}(X)$ language, we postulate the constant $\{\cdot\}$, and the families of constants $a:\cdot$ and $\cdot\{a = \cdot\}$, with semantics as defined in Fig. 2. Thus, we define the constants of $\text{pml}$, along with their initial type bindings, in Fig. 9.
\[
\{\cdot\} : \forall \alpha. \alpha \rightarrow \{\partial \alpha\}
\]
\[
\cdot\{a = \cdot\} : \forall \alpha_1 \alpha_2 \beta. \{a : \alpha_1 ; \beta\} \rightarrow \alpha_2 \rightarrow \{a : \alpha_2 ; \beta\}
\]
\[
B : \{B+, \emptyset\}
\]
\[
\bar{B} : \{B-, \omega\}
\]
\[
\exists b : \forall \beta. \{b+, \beta\} \rightarrow \{b+, \beta\}
\]
\[
\wedge : \forall \beta_1 \beta_2 \beta_3[C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}
\]
\[
\text{where } C = \begin{cases} & \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ & \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{cases}
\]
\[
\vee : \forall \beta_1 \beta_2 \beta_3[C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}
\]
\[
\text{where } C = \begin{cases} & \text{if } + \leq \beta_1 \text{ then } \omega \leq \beta_3 \\ & \text{if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{cases}
\]
\[
\exists : \forall \beta_1 \beta_2 \beta_3[C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}
\]
\[
\text{where } C = \begin{cases} & \text{if } + \leq \beta_2 \text{ then } \emptyset \leq \beta_3 \\ & \text{if } - \leq \beta_2 \text{ then } \beta_1 \leq \beta_3 \end{cases}
\]
\[
\exists_\gamma : \forall \alpha \beta \gamma[C]. \{b \gamma, \beta\} \rightarrow (\{b+, \beta_1\} \rightarrow \alpha_1) \rightarrow (\{b-, \beta_2\} \rightarrow \alpha_2) \rightarrow \alpha
\]
\[
\text{where } C = \begin{cases} & \text{if } + \leq \gamma \text{ then } \beta \leq \beta_1 \wedge \text{if } - \leq \gamma \text{ then } \beta \leq \beta_2 \\ & \text{if } + \leq \gamma \text{ then } \alpha_1 \leq \alpha \wedge \text{if } - \leq \gamma \text{ then } \alpha_2 \leq \alpha \end{cases}
\]

Fig. 9. Constants and initial type bindings for pml

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:

\[
B_1 = \{b_1, b_2, b_3\}
\]
\[
B_2 = \{b_1, b_2, b_4\}
\]

Suppose then that we wish to type the expression \(B_1 \land B_2\), using the unification-based constraint system RS*\(^\ominus\). Given the typing for \(\land\) defined in Fig. 9, the variables \(\beta_1\) and \(\beta_2\) will be unified with the types of the contents of \(B_1\) and
$B_2$, respectively:

$$\beta_1 = (b_1+, b_2+, b_3+, \emptyset)$$
$$\beta_2 = (b_1+, b_2+, b_4+, \emptyset)$$

Additionally, $\beta_3$ 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, the constraint $C$ in the type of $\wedge$ can be expanded as follows:

$$C = \begin{cases} 
\text{if } - \leq + \text{ then } - \leq \gamma_1 \land \text{ if } + \leq + \text{ then } + \leq \gamma_1 \\
\land \text{ if } - \leq + \text{ then } - \leq \gamma_2 \land \text{ if } + \leq + \text{ then } + \leq \gamma_2 \\
\land \text{ if } - \leq + \text{ then } - \leq \gamma_3 \land \text{ if } + \leq + \text{ then } - \leq \gamma_3 \\
\land \text{ if } - \leq - \text{ then } - \leq \gamma_4 \land \text{ if } + \leq - \text{ then } + \leq \gamma_4 \\
\land \text{ if } - \leq \emptyset \text{ then } \emptyset \leq \beta \land \text{ if } + \leq \emptyset \text{ then } \emptyset \leq \beta 
\end{cases}$$

This expansion will force the following unification:

$$\beta_3 = (b_1+, b_2+, b_3-, b_4-, \emptyset)$$

or

$$\beta_3 = (b_1+, b_2+, \emptyset)$$

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

### 4.2 Type soundness for pml

Given the previous development, we may now define the type systems for pml. Specifically, these are denoted $\mathcal{S}^\leq$ and $\mathcal{S}^\leq$ and obtained from extending $\text{HM}(\text{RS}^\leq)$ and $\text{HM}(\text{RS}^\leq)$, 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 $\text{HM}(X)$ in either of $\mathcal{S}^\leq$, we demonstrate the requisite $\delta$-typability lemma, which means that the initial type bindings for the pml constants are sound:

**Lemma 4.2** pml is $\delta$-typable in $\mathcal{S}^\leq$.

The proof is straightforward, and is omitted here for brevity. Given this lemma, the soundness of $\text{RS}^\leq$, and results demonstrated in [13], we immediately obtain type soundness for pml in both type systems:

**Theorem 4.3** (pml Type Soundness) If $e$ is a pml expression which is well-typed in $\mathcal{S}^\leq$, then $e$ does not go wrong.
In addition to this result, the systems $S^{rel}$ enjoy the benefits of type inference: $\text{HM}(X)$ provides a type inference algorithm modulo constraint solution \cite{1}, and a row type and conditional constraint solution algorithm exists and has been proven correct \cite{2}.

A consequence of Theorem 4.3 is that certain pml runtime optimizations may be effected. For example, this result implies that all membership checks $\exists_b$ 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 $\leadsto$ be defined as $\rightarrow$, but with the memcheck rule redefined as $B \triangleright b \leadsto B$; that is, no runtime membership checks are performed. Suppose $e$ is well typed; then $e \leadsto^* v$ iff $e \rightarrow^* v$.

# 5 Characterizing pml: 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 product-coproducts. The asymmetric nature of programs and continuations causes the duality between records and variants \cite{14} to be imperfect: while the type-indexed rows of \cite{12} 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, reinforcing our argument that pml 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):

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

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:

\[
\text{freeze } e \triangleq \lambda x. e \quad \text{where } x \text{ not free in } e
\]

\[
\text{thaw } e \triangleq e \{\} \quad \text{fail } \triangleq \emptyset \ni b
\]

\[
[b] = \lambda r. (\text{thaw } r.a_b)
\]

\[
[\text{match } e_1 \text{ with } b \rightarrow e_2] = [e_1]((\text{freeze fail})\{ a_b = \text{freeze } [e_2] \})
\]

\[
[\text{match } e_1 \text{ with } b \rightarrow e_2 \mid \_ \rightarrow e_3] = [e_1]((\text{freeze } [e_3])\{ a_b = \text{freeze } [e_2] \})
\]

\[
\vdots
\]

And so on trivially for the other functional language constructs. Since pml 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 labelling system that can be expressed via enumerations.

5.2 Encoding sets with enumerations

Since sets in pml 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 \(\exists_b\) 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 \(\text{fresh}_b()\) during the encoding:

\[
\text{succeed } \triangleq \lambda x. ()
\]

\[
\text{fail } \triangleq \lambda x. \text{match } x \text{ with } \text{fresh}_b() \rightarrow ()
\]
\[
\{ [b_1, \ldots , b_n] \} = \lambda f . \lambda g . \lambda x . \text{match } x \text{ with } b_1 \rightarrow f(x) \mid \\
\vdots \\
    b_n \rightarrow f(x) \mid \_ \rightarrow g(x) \\
\{ [\{b_1, \ldots , b_n\}] \} = \lambda f . \lambda g . \lambda x . \text{match } x \text{ with } b_1 \rightarrow g(x) \mid \\
\vdots \\
    b_n \rightarrow g(x) \mid \_ \rightarrow f(x) \\
[\land] = \lambda s_1 . \lambda s_2 . \lambda f . \lambda g . \ s_1 (s_2 f \, g) \ g \\
[\lor] = \lambda s_1 . \lambda s_2 . \lambda f . \lambda g . \ s_1 f \ (s_2 f \, g) \\
[\exists] = \lambda s_1 . \lambda s_2 . \lambda f . \lambda g . \ s_1 (s_2 g \, f) \ g \\
[\forall b] = \lambda s . \, \text{succeed fail}'b \\
[? b] = \lambda s . \lambda y . \lambda z . \ s \ (\lambda x . y)(\lambda x . z)'b \\
\vdots
\]

And so on trivially for the other functional language constructs. Not only does this suggest an alternate implementation for sets; it also suggests an alternate implementation of set types, since polymorphic variant types have been previously studied \[29\] and exist in e.g. OCaml. However, the types in such an implementation would not be nearly as readable as set types (see Sect. \[6\]).

\section{Applications}

In this section we discuss several applications of pml as an implementation language for higher-level languages, demonstrating its usefulness as a uniform base for this purpose; in particular, the precise static analysis of pml facilitates run-time optimizations in these implementations.

\subsection{Stack inspection}

In \[8\] the authors implement the so-called security-passing-style transformation from a language with stack inspection security, called \(\lambda_{\text{sec}}\), into a less general form of pml, called \(\lambda_{\text{set}}\). 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 \(\lambda_{\text{sec}}\)-to-\(\lambda_{\text{set}}\) transformation provides an indirect static analysis for \(\lambda_{\text{sec}}\), 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 $\lambda_{sec}$, providing insight into its form and greatly easing proof of its soundness.

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 $A$. Individual privileges $r$ are explicitly activated via expressions of the form $\text{enable } r \in e$, and their activation may be checked via expressions of the form $\text{check } r \text{ then } e$. 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 $\text{PrintPriv}$ privilege before printing:

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

Printing could then be accomplished in an environment with $\text{PrintPriv}$ enabled, by using safeprint in code owned by principals locally authorized for $\text{PrintPriv}$. The stack inspection algorithm prevents any code owned by principals not authorized for $\text{PrintPriv}$ 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 $s$:

$$[\text{enable } r \in e]_p = \text{let } s = s \lor (\{r\} \cap A(p)) \text{ in } [e]_p$$

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 $s$:

$$[\text{check } r \text{ then } e]_p = \text{let } s = s \lor r \text{ in } [e]_p$$

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:

$$[p'.e]_p = \text{let } s = s \land A(p') \text{ in } [e]_{p'}$$

A significant feature of this translation is that the $\lambda_{sec}$ type system—subsumed in the pml 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.
6.0.2 Object confinement

In [14] 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, utilizing most of the language and type features presented here. However, pml 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 here—but this is a minor detail, since the presentation of HM(X) in [13] contains state as part of the core calculus. As is the case for λsec, the pop-to-pml transformation also drives the development of a direct type system for pop, and eases its soundness proof, while the precise pml 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 \(\varphi\), 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 \(\partial\). For example, a file object \(o\) may be defined as follows, which is read/write in its own domain, but read-only otherwise:

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

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 transformation are then defined as follows. The transformation of interfaces \(\varphi\) is denoted \(\hat{\varphi}\), and uses records with default values in the image:

\[
\{d_1 \mapsto \iota_1, \ldots, d_n \mapsto \iota_n, \partial \mapsto \iota\} = \{\iota\}\{d_1 = \iota_1\} \ldots \{d_n = \iota_n\}
\]

Objects are transformed roughly as follows\(^3\), where advanced features such

\(^3\) In these transformations, we use the following syntactic sugar:

\[
\{m_1 = e_1, \ldots, m_n = e_n\} \triangleq \{\emptyset\}\{m_1 = e_1\} \cdots \{m_n = e_n\}
\]
\[
e_1; e_2 \triangleq \text{let } x = e_1 \text{ in } e_2 \quad x \text{ not free in } e_2
\]
as the treatment of “self” are omitted here for brevity:

\[
[[m_1(x) = e_1, \ldots, m_n(x) = e_n] \cdot d \cdot \varphi]_d' = \{\text{obj} = \{m_1 = \lambda x.\{e_1\}_d, \ldots, m_n = \lambda x.\{e_n\}_d\}, \text{ifc} = \varphi\}
\]

Then, method selects are encoded so that access rights are always verified:

\[
[e_1.m(e_2)]_d = \text{let } c_1 = [e_1]_d \text{in}
\]
\[
c_1.\text{ifc}.d \ni m; \quad (c_1.\text{obj}.m)([e_2]_d)
\]

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

6.0.3 Flow types

In [3] and [16], the authors develop flow type systems which use source and sink labels on functions and application points, respectively, along with type annotations. The type annotations conservatively approximate all sinks to which a particular source flows, and all sources that flow to a particular sink. A type checking algorithm verifies these annotations. Here, we provide an encoding of this analysis using the pml language. A benefit of this encoding is that flow information is inferred by the pml type system, while the flow types of [3,16] presuppose some unspecified analysis that provides flow annotations.

It should be pointed out that the systems $\mathcal{S}^{rel}$ do not have the expressiveness of the flow types of [16], which achieve a polyvariant analysis through the use of union and intersection types; in particular, their system specifies the type that polyvariant functions are instantiated to at specific application points, information that benefits compiler optimization. However, our system is as expressive as any in [3]. Furthermore, the scheme presented here can serve as the foundation for a more expressive polyvariant flow type inference, by incorporation of the type analysis in [15]. This would provide the expressiveness of polyvariance without affecting the type language defined here, nor the definition of the encoding.

Our encoding of flow types is defined as follows. First, all programs (top-level expressions) $e$ are wrapped in a function whose parameter flow is an abstracted “flow table”, a table mapping sources to sets of sinks, the idea being that the encoding enforces a safe approximation of program flow information.
in the type of flow. This is the top level encoding \([e]\):

\[ [e] = \lambda flow.[e] \]

The subexpressions \(e'\) of \(e\) are then transformed by the encoding \([e']\). Every function is associated with a fresh source label \(a\), returned by the function \(\text{fresh}_a()\), during the encoding. Each function is also assigned an extra parameter \(u\), which is used to pass in a function at every application point, checking that the sink label \(b\) associated with that point is assigned to the set of specified sinks for \(a\) in flow. Application points are likewise assigned fresh labels during the encoding by calls to \(\text{fresh}_b()\):

\[ [\lambda x.e] = \lambda u.\lambda x.((flow.\text{fresh}_a()); [e]) \]
\[ [e_1e_2] = [e_1](\lambda x.(x \ni \text{fresh}_b()))[e_2] \]

To clarify how the encoding works, we give the following example.

**Example 6.1** Let the source expression \(e\) be defined as follows:

\[
\begin{align*}
\text{let } f_1 &= \lambda x.x \text{in} \\
\text{let } f_2 &= \lambda f.f\,1 \text{in} \\
f_1\,1; f_1\text{true}; f_2\,f_1
\end{align*}
\]

Then, the encoding \([e]\) is as follows, assuming a trivial extension of pml with integers and booleans; for brevity, and to retrieve the syntactic style of [16], we define the macros \(\@ b = \lambda x.(x \ni b)\) and \(\lambda^a.x.e = \lambda u.\lambda x.((u.(flow.a); e))\):

\[
\begin{align*}
\lambda flow. \\
\text{let } f_1 &= \lambda^a_1.x.x \text{in} \\
\text{let } f_2 &= \lambda^a_2.f.f\@ b_1.1 \text{in} \\
f_1\@ b_2.1; f_1\@ b_3\text{true}; f_2\@ b_4.f_1
\end{align*}
\]

In the unification-based system \(S^=\), the inferred type of flow for \([e]\) in this case is:

\(flow : \{a_1 : \{b_1+, b_2+, b_3+, b_1\} ; a_2 : \{b_4+, \beta_2\} ; \beta_3\}\)

Note that the type specifies the correct sinks for each of the sources in the program. Note also that the converse source-per-sink information can be obtained by inspection of the type of flow. Finally, since the flow checks are verified by the type system, they can be eliminated after static analysis, as a consequence of Proposition 4.4.

18
7 Conclusion

In this paper we have defined the pml 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 \( \text{HM}(X) \), which provides an easy method for proof of type soundness and definition of type inference.

We also describe applications of pml as an implementation language for other systems, including a language incorporation stack inspection, and a language for modelling object confinement mechanisms. Via these implementations, the precise pml type discipline provides a static analysis of dynamic properties for these systems, allowing runtime optimizations. Additionally, we describe an alternative encoding of flow types [3][10] that provides an inference method for flow information, rather than requiring this information to pre-exist in type annotations. These diverse applications also suggest that there may be other useful applications of pml as an implementation language for higher-level systems. Future work includes an incorporation of the polyvariant type inference analysis of [15] into the pml type system, which will increase its usefulness as an inference method for polyvariant flow types [16].

References


