

Set Types and Applications

Christian Skalka and Scott Smith
The Johns Hopkins University

Topics: the language pml

We define pml , a functional language extended with:

- *records* in the style of Projective ML (*Rémy, CLFP92*)
- *sets* of atomic elements, set operations

The language includes an accurate type system that captures dynamic properties of records and sets:

- *row types* accurately describe contents of records and sets
- *conditional constraints* accurately describe behavior of set operations

Language and type system defined as an *instance of* $\text{HM}(X)$

Topics: applications of pml

The language pml serves as a useful *target language* for semantics-preserving transformation of various source languages:

- an *intermediate* language
- a basis for the *derivation* of source language type systems

In either case, the pml type system provides a static analysis enabling run-time optimizations for source languages

Language details

To define `pml` as an instance of $\text{HM}(X)$, we *populate the set* `Const` with record constructors and destructors:

- elevation: $\{\cdot\}$, where $\{v\}$ is a record with default value v
- modification: $\cdot\{a = \cdot\}$, where $v\{a = v'\}$ contains v' in a field
- projection $\cdot.a$, where $v.a$ selects contents of a field from v

and a language of sets:

- sets of atomic elements: $B \subseteq \mathcal{L}_b$, with $\mathcal{L}_b = \{b_1, b_2, \dots\}$
- cosets: \bar{B} , where $\bar{B} = \mathcal{L}_b - B$
- set operations: membership check \ni_b , union \vee , intersection \wedge , difference \ominus

pml examples

Dynamic behavior is described via an operational semantics relation \rightarrow :

$$\textit{intersection:} \quad B_1 \wedge B_2 \rightarrow B_1 \cap B_2$$

$$\textit{union:} \quad B_1 \vee B_2 \rightarrow B_1 \cup B_2$$

$$\textit{membership:} \quad B \ni b \rightarrow B \quad \text{if } b \in B$$

$$\textit{skip project:} \quad v\{a' = v'\}.a \rightarrow v.a \quad a' \neq a$$

$$\textit{project:} \quad v\{a = v'\}.a \rightarrow v'$$

⋮

These *define the function* δ in the instantiation of $\text{HM}(X)$.

Types for pml

The pml type system is defined by extending the $HM(X)$ type and constraint language with the system RS (**R**ows and **S**ets):

- a *type and constraint language* comprising row types and conditional constraints
- a *standard interpretation* in a model, specifying behavior of conditional constraints
- *initial type bindings* Δ for records, sets and associated operations

The system RS

We extend the basic $\text{HM}(X)$ system with row types and *presence constructors*:

$$\begin{array}{lll}
 \zeta & ::= & \alpha, \beta, \dots \mid (b : \tau; \zeta) \mid \partial\tau & \text{rows} \\
 c & ::= & + \mid - & \text{constructors} \\
 \tau & ::= & \alpha, \beta, \dots \mid \tau \rightarrow \tau \mid \{\zeta\} \mid c & \text{types}
 \end{array}$$

These types describe sets by asserting which elements are present (+) and which are absent (-); letting $B = \{b_1, b_2\}$:

$$\begin{array}{l}
 B : \{b_1 : +; b_2 : +; \partial-\} \\
 B : \{b_1 : +; b_2 : +; b_3 : -; \partial-\}
 \end{array}$$

We define the following syntactic sugar for types:

$$(b : \tau; \zeta) \triangleq (b\tau, \zeta) \qquad \partial- \triangleq \emptyset \qquad \partial+ \triangleq \omega$$

Hence:

$$B : \{b_1+, b_2+, \emptyset\}$$

Conditional constraints

The RS language of constraints extends the $HM(X)$ constraint language with *conditional constraints**:

$$C ::= \dots \mid \text{if } c \leq \tau \text{ then } \tau' \leq \tau'' \quad \textit{constraints}$$

The behavior of conditional constraints is specified by the RS interpretation, and includes an “intuitive” interpretation:

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

*Pottier, *Nord. J. Comp.*, Nov. 2000

Conditional constraints

Conditional constraints are also equipped with a more complex interpretation:

$$\frac{\forall b \in \mathcal{L}_b . (c \leq \rho(\zeta)(b) \Rightarrow \rho(\zeta')(b) \leq \rho(\zeta'')(b))}{\rho \vdash \text{if } c \leq \zeta \text{ then } \zeta' \leq \zeta''}$$

This interpretation is used for an accurate description of the behavior of set operations:

$$\begin{aligned} \wedge : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\} \\ \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ \quad \wedge \text{ if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

Conditional constraint example

$$\begin{aligned} \wedge & : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\} \\ & \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ & \quad \wedge \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

Define \leq as $=$; then to type the expression $\{b_1, b_2\} \wedge \{b_2, b_3\}$:

$$\beta_1 = \{b_1+, b_2+, b_3-, \emptyset\}$$

$$\beta_2 = \{b_1-, b_2+, b_3+, \emptyset\}$$

$$\beta_3 = \{b_1\gamma_1, b_2\gamma_2, b_3\gamma_3, \beta\}$$

Conditional constraint example

$$\begin{aligned} \wedge : \forall \beta_1 \beta_2 \beta_3 [C]. \{ \beta_1 \} \rightarrow \{ \beta_2 \} \rightarrow \{ \beta_3 \} \\ \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ \quad \wedge \text{if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

The interpretation of constraints will force the following “splitting”:

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

Conditional constraint example

$$\begin{aligned} \wedge : \forall \beta_1 \beta_2 \beta_3 [C]. \{ \beta_1 \} \rightarrow \{ \beta_2 \} \rightarrow \{ \beta_3 \} \\ \text{where } C = \quad \text{if } - \leq \beta_1 \text{ then } \emptyset \leq \beta_3 \\ \quad \wedge \text{ if } + \leq \beta_1 \text{ then } \beta_2 \leq \beta_3 \end{aligned}$$

This splitting will force the following unification:

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

or

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

This describes the contents of $\{b_1\}$, and $\{b_1, b_2\} \wedge \{b_2, b_3\} \rightarrow^* \{b_2\}$.

Other pml initial bindings

$$\exists b : \forall \beta. \{b+, \beta\} \rightarrow \{b+, \beta\}$$

$$\vee : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}$$

$$\text{where } C = \quad \text{if } + \leq \beta_1 \text{ then } \omega \leq \beta_3$$

$$\quad \wedge \text{ if } - \leq \beta_1 \text{ then } \beta_2 \leq \beta_3$$

$$\ominus : \forall \beta_1 \beta_2 \beta_3 [C]. \{\beta_1\} \rightarrow \{\beta_2\} \rightarrow \{\beta_3\}$$

$$\text{where } C = \quad \text{if } + \leq \beta_2 \text{ then } \emptyset \leq \beta_3$$

$$\quad \wedge \text{ if } - \leq \beta_2 \text{ then } \beta_1 \leq \beta_3$$

Type safety for pml

Note that the type binding

$$\ni b : \forall \beta. \{b+, \beta\} \rightarrow \{b+, \beta\}$$

requires that b be an element of B for any expression $B \ni b$, so that any operationally unsafe membership check is not well-typed.

- *Type safety* implies that *optimizations* may be effected; run-time membership checks may be eliminated

NB: Due to the use of $\text{HM}(X)$, type safety is obtained by a simple proof of soundness of initial bindings (δ -typability)

Applications: stack inspection

In recent work*, a type system for a model of Java JDK1.2 architecture is developed by *transformation* into a restricted form of pml called λ_{set}

Java JDK1.2 model, called λ_{sec} , comprises functional core, plus additions for modelling *stack inspection* security:

- access control lists: \mathcal{A}
- signed expressions: $p.e$
- privilege enabling: check r then e
- privilege checking: enable r in e

*Pottier, Skalka and Smith ESOP01

Applications: stack inspection

In the Java JDK1.2 model, security is literally *maintained on call stack*; security checks initiate stack search for valid privilege activation.

In λ_{sec} -to- λ_{set} transformation, privilege activations are maintained in privilege set s which are *passed down the stack*:

$$\llbracket \lambda x.f \rrbracket_p = \lambda x.\lambda s.\llbracket f \rrbracket$$

$$\llbracket \text{enable } r \text{ in } e \rrbracket_p = (\text{let } s = s \vee (\{r\} \cap \mathcal{A}(p)) \text{ in } \llbracket e \rrbracket_p)$$

$$\llbracket \text{check } r \text{ then } e \rrbracket_p = (\text{let } _ = s \ni r \text{ in } \llbracket e \rrbracket_p) \quad (*\textit{statically enforced}*)$$

$$\llbracket p'.e \rrbracket_p = (\text{let } s = s \wedge \mathcal{A}(p') \text{ in } \llbracket e \rrbracket_{p'})$$

- transformation allows elimination of dynamic security checks

Applications: stack inspection

The pml language therefore yields significant benefits for stack inspection model:

- language optimizations may be effected by using pml as *implementation* language, pml types as *indirect* analysis

A *direct* static analysis for λ_{sec} may also be developed via transformation into pml:

- form of direct types based on types of transformed terms
- direct type safety easy to prove, by syntactic correspondance with indirect analysis

Applications: object confinement

In recent work*, the OO language `pop` is developed for enforcement of object confinement:

- objects are explicitly assigned *domain* names d
- objects are endowed with *interfaces* φ , which specify access rights via mappings $d \mapsto \iota$
- access authorization enforced by *dynamic* checks

For example, a file object that is read/write locally but read-only elsewhere may be defined as follows, where ∂ is the *default* domain:

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

*Skalka and Smith, FCS02, To appear

Applications: object confinement

As for λ_{sec} , a static analysis for pop may be developed by transformation into pml.

The transformation of interfaces φ is denoted $\hat{\varphi}$, and uses records with default values in the image:

$$\{d_1 \mapsto \iota_1, \dots, \widehat{d_n \mapsto \iota_n}, \partial \mapsto \iota\} = \{\iota\} \{d_1 = \iota_1\} \dots \{d_n = \iota_n\}$$

The transformation of objects uses both records and sets in the image:

$$\begin{aligned} & \llbracket [m_1(x) = e_1, \dots, m_n(x) = e_n] \cdot d \cdot \varphi \rrbracket_{d'} \\ & = \\ & \{\text{obj} = \{m_1 = \lambda x. \llbracket e_1 \rrbracket_d, \dots, m_n = \lambda x. \llbracket e_n \rrbracket_d\}, \text{ifc} = \hat{\varphi}\} \end{aligned}$$

Applications: object confinement

The transformation of object selection implements access authorization as a set membership check:

$$\llbracket e_1.m(e_2) \rrbracket_d = \text{let } c_1 = \llbracket e_1 \rrbracket_d \text{ in} \\ c_1.\text{ifc}.d \ni m; \\ (c_1.\text{obj}.m)(\llbracket e_2 \rrbracket_d)$$

- pml type system statically enforces security in the transformation
- a *direct* static analysis for pop is easily developed on the foundation of pml type system

Applications: flow types

*Flow types** promote type-directed compiler optimizations by use of *source* and *sink* labels:

- functions are assigned source label, application points are assigned sink label
- functions are additionally labeled with the sink labels to which they flow
- application points are labeled with the sources that flow to them
- type *checking* algorithm verifies labelling

*Heintze, SAS95 Wells et al., JFP 200X, To appear

Applications: flow types

While flow types are useful, previous presentations *presuppose* an unspecified algorithm that provides flow type annotations.

Monomorphic flow types may be encoded using pml as an intermediate language:

- source labels encoded as record labels
- sink labels encoded as set elements
- set membership checks at application points, with respect to *abstract* “flow table” record, encodes source-to-sink flow information

Using this scheme, flow types may be accurately *inferred*.

Conclusion, future work

We have presented the pml language:

- includes language of extensible records, sets, and set operations
- includes a type system than uses conditional constraints to accurately describe behavior of language features

The pml language may be used to provide direct and indirect static analyses for languages with security features:

- stack inspection
- object confinement

Conclusion, future work

The language pml may also be used as an intermediate language to encode monomorphic flow type inference.

- in all cases, the pml type system promotes run-time optimizations for source languages

Future work:

- incorporation of *polyvariant* analysis (Smith and Wang, ESOP00) to provide a more expressive flow type inference method
- implementation language for other label-based source languages?

<http://www.cs.jhu.edu/~ces/work.html>