# Verifying Pointer and String Analyses with Region Type Systems

Lennart Beringer[1], Robert Grabowski[2], and Martin Hofmann[2]

[1] Department of Computer Science, Princeton University,
35 Olden Street, Princeton 08540, New Jersey
eberinge@cs.princeton.edu

[2] Institut für Informatik, Ludwig-Maximilians-Universität,
Oettingenstrasse 67, D-80538 München, Germany
{robert.grabowski,martin.hofmann}@ifi.lmu.de

**Abstract.** Pointer analysis statically approximates the heap pointer structure during a program execution in order to track heap objects or to establish alias relations between references, and usually contributes to other analyses or code optimizations. In recent years, a number of algorithms have been presented that provide an efficient, scalable, and yet precise pointer analysis. However, it is unclear how the results of these algorithms compare to each other semantically.

In this paper, we present a general region type system for a Java-like language and give a formal soundness proof. The system is subsequently specialized to obtain a platform for embedding the results of various existing context-sensitive pointer analysis algorithms, thereby equipping the computed relations with a common interpretation and verification. We illustrate our system by outlining an extension to a string value analysis that builds on pointer information.

## 1 Introduction

Pointer (or points-to) analysis is a static program analysis technique that determines an over-approximation of possible points-to relations that occur during the execution of a program. More precisely, it chooses an abstraction of the pointers and references, and computes which pointers may possibly point to which data. A conservative approximation of this structure is also an alias analysis, as the computed points-to relation directly includes the information which pointers may point to the same object. A pointer analysis is often used for compiler optimizations, but may also serve as the basis for other analyses, such as the computation of possible string values in order to prevent string-based security holes.

There exists a large number of pointer analysis algorithms [1,2,3] for different languages. Each algorithm faces the trade-off between precision and efficiency of the analysis: it should choose the right abstractions in order to produce as much useful information as possible while at the same time being able to process large code bases in a reasonable time. These analyses have different techniques, implementations, and complexities. Especially BDD-based algorithms [4,5] have been shown to be very efficient and precise at the same time.

While several of these analyses also consider soundness, it appears that there does not yet exist a uniformly agreed-upon formal framework that encompasses the interpretations of at least a substantial subset of the analyses. We argue that such a unifying treatment is important, for theoretical as well as pragmatic and practical reasons. First, it is a basis for fair comparisons regarding the precision, flexibility or expressivity of different analyses, the theoretical complexity of the associated algorithms, and their experimental evaluation on common benchmarks. Second, once the analyses agree on the formal property they guarantee, we can safely replace one analysis by another in a compiler or verification tool. Third, a uniform guarantee provides the basis for the formal verification of security-relevant properties that rely on pointer analysis results, as is required in proof-carrying code scenarios.

The first purpose of the present paper is to provide such a framework for Java-like languages, given by a hierarchy of region-based type systems for a language in the style of Featherweight Java [6]. Uniformity (i.e. semantic agreement) is guaranteed by equipping the bottom-most layer in the hierarchy with a formal interpretation and soundness result, and by deriving the higher levels by specializing this bottom-most layer to move towards calculi representing concrete analyses.

Second, we demonstrate that a number of existing pointer analyses for object-oriented programs are based on abstraction disciplines that arise as specializations of a generic parametrized refinement of our base-level type system. We focus on disciplines that specify the abstraction of references and execution points [7], and therefore enable different forms of field-sensitive and context-sensitive analyses. For example, objects may be abstracted to their allocation sites and their class, and execution points may be abstracted by receiver-object or call-site stack contexts.

As one moves higher in the hierarchy, different choices for these abstractions regarding expressivity arise, corresponding to the above abstraction disciplines, and decidability issues become more prominent, in particular the question of algorithmic type checking. Our third contribution consists of showing how the parametrized type system can be reformulated algorithmically to yield a type checking algorithm. Thanks to the hierarchical structure of our framework, we thus immediately obtain algorithms for automatically validating the correctness of the results of concrete analyses, as long as the results have been interpreted in the framework by instantiating its parameters. As we only consider the results, our verification is also independent of implementation details of concrete analysis algorithms.

Finally, we apply our framework to the analysis of string values, in order to lay the foundations for eliminating SQL injections and cross-site scripting attacks. We extend the language to a simple string model, and outline how data flow analyses for possible string values that build on pointer analyses [8,9] can be verified with the correspondingly extended type system.

*Related work.* We include concepts of context-sensitivity to distinguish different analyses of the same method implementation. In particular, $k$-CFA [10] is a higher-order control flow analysis where contexts are call stack abstractions of finite length $k$. The $k$-CFA mainly addresses control flows in functional languages, where functions are first-class values. It requires a combination of value (data flow) analysis and control flow analysis to approximate the possible lambda abstractions that an expression may evaluate to. The similar dynamic dispatch problem for methods in object-oriented languages is easier to solve, as the possible implementation targets of a method invocation can be retrieved from the class information. $k$-CFA has been extended with polyvariant types for functions [11], such that different types can be used for the function at different application sites. In our type system, we borrow this concept of polyvariance.

Hardekopf and Lin [12] transform variables that are not address-taken into SSA form, such that running a flow-insensitive analysis on these converted variables has the effect of running a flow-sensitive analysis on the original variables. Our framework assumes SSA-transformed code input, presented in the form of a functional program. Identifying opportunities for SSA transformation, which is a central concern of [12], can thus be seen as a preprocessing phase for our framework to apply.

This paper uses "regions" in the sense of Lucassen and Gifford [13], i.e. as representations of disjoint sets of memory locations. Equivalently, regions thus partition or color the memory. In literature, this disjointness is used to show that certain memory manipulations do not influence other parts of a program, in order to e.g. show semantic equivalences [14], to enable a safe garbage collection [15], or to infer properties for single objects by tracking unique objects in a region [16,17]. In the pointer analysis setting presented here, regions are simply seen as abstract memory locations that summarize one or more concrete locations, thereby helping to discover may-alias relations. We do not consider uniqueness or must-alias information, and do not aim to justify garbage collection.

Paddle [18] and Alias Analysis Library [19] are implementation frameworks that embed concrete pointer analyzers by factoring out different parameters, as is done here. However, the works do not aim at a formal soundness result. Indeed, they embed the *algorithms* into a common *implementation* framework, while our type system embeds the *results* of such algorithms into a common *semantic* framework.

*Synopsis.* The next section introduces the FJEU language and its semantics. We introduce the base-level region type system and the soundness proof in section 3. In section 4, we specialize the type system to a parametrized version, such that abstraction principles found in pointer analyses can be modeled explicitly as instantiations of the parameters, and outline a type-checking algorithm. Finally, section 5 extends the system, such that results of a string analysis based on pointer analysis can also be verified.

## 2 Featherweight Java with Updates

We examine programs of the language FJEU [20], a simplified formal model of the sequential fragment of Java that is relevant for pointer analysis. FJEU extends Featherweight Java (FJ) [6] with attribute updates, such that programs may have side effects on a heap. Object constructors do not take arguments, but initialize all fields with *null*, as they can be updated later. Also, the language adds `let` constructs and conditionals to FJ. For a small example, please refer to the full version of this paper [21], which shows a small list copy program in Java and a corresponding implementation in FJEU.

### 2.1 Preliminaries

We write $\mathcal{P}(X)$ for the set of all subsets of $X$. The notations $A \to B$ and $A \rightharpoonup B$ stand for the set of total and partial functions from $A$ to $B$, respectively. We write $[x \mapsto v]$ for the partial function that maps $x$ to $v$ and is else undefined. The function $f[x \mapsto v]$ is equal to $f$, except that it returns $v$ for $x$. Both function notations may be used in an indexed fashion, e.g. $f[x_i \mapsto v_i]_{\{1,\dots,n\}}$, to define multiple values. In typing rules, we sometimes write $x : v$ and $f, x : v$ corresponding to the above notation. Finally, we write $\overline{a}$ for a sequence of entities $a$.

### 2.2 Syntax

The following table summarizes the (infinite) abstract identifier sets in the language, the meta-variables we use to range over them, and the syntax of FJEU expressions:

|  |  |
|---|---|
| variables: $x, y \in \mathcal{X}$ | classes: $C, D, E \in \mathcal{C}$ |
| fields: $f \in \mathcal{F}$ | methods: $m \in \mathcal{M}$ |

$$\begin{aligned}\mathcal{E} \ni e ::= {} & \texttt{null} \mid x \mid \texttt{new}\ C \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid x.f \mid x.f := y \mid x.m(\overline{y}) \mid \\ & \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{if}\ x = y\ \texttt{then}\ e\ \texttt{else}\ e\end{aligned}$$

To keep our calculus minimal and focused on pointer alias analysis, we omit primitive data types such as integers or booleans. However, such data types and their operations can easily be added to the language. We also omit type casts found in FJ, as they do not provide new insights to pointer analysis. In order to simplify the proofs, we require programs to be in let normal form. The somewhat unusual conditional constructs for dynamic class tests and value equality are included to have reasonable if-then-else expressions in the language while avoiding the introduction of booleans.

An FJEU program is defined by the following relations and functions:

$$\begin{aligned}\text{subclass relation:}\quad & \prec \in \mathcal{P}(\mathcal{C} \times \mathcal{C}) \\ \text{field list:}\quad & \mathit{fields} \in \mathcal{C} \to \mathcal{P}(\mathcal{F}) \\ \text{method list:}\quad & \mathit{methods} \in \mathcal{C} \to \mathcal{P}(\mathcal{M}) \\ \text{method table:}\quad & \mathit{mtable} \in \mathcal{C} \times \mathcal{M} \rightharpoonup \mathcal{E} \\ \text{FJEU program:}\quad & P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})\end{aligned}$$

FJEU is a language with nominal subtyping: $D \prec C$ means $D$ is an immediate subclass of $C$. The relation is well-formed if it is a tree successor relation; multiple inheritance is not allowed. We write $\preceq$ for the reflexive and transitive hull of $\prec$. The functions *fields* and *methods* describe for each class $C$ which fields and method objects of that class have. The functions are well-formed if for all classes $C$ and $D$ such that $D \preceq C$, $\mathit{fields}(C) \subseteq \mathit{fields}(D)$ and $\mathit{methods}(C) \subseteq \mathit{methods}(D)$, i.e. classes inherit fields and methods from their superclasses. A method table *mtable* gives for each class and each method identifier its implementation, i.e. the FJEU expression that forms the method's body. To simplify the presentation, we assume that formal argument variables in the body of a method $m$ are named $x_1^m$, $x_2^m$, etc., abbreviated to $\overline{x^m}$, besides the implicit and reserved variable $\mathit{this}$. All free variables of an implementation of $m$ must be from the set $\{\mathit{this}, x_1^m, x_2^m, \ldots\}$. A method table is well-formed if $\mathit{mtable}(C, m)$ is defined whenever $m \in \mathit{methods}(C)$. In other words, all methods declared by *methods* must be implemented, though the implementation may be overridden in subclasses for the same number of formal parameters. In the following, we assume a fixed FJEU program $P$ whose components are all well-formed.

## 2.3 Semantics

A state consists of a store (variable environment or stack) and a heap (memory). Stores map variables to values, while heaps map locations to objects. An object consists of a class identifier and a valuation of its fields. The only kind of values in FJEU are locations and *null* references.

$$\begin{array}{rlrl} \text{locations:} & l \in \mathcal{L} & \text{stores:} & s \in \mathcal{X} \rightharpoonup \mathcal{V} \\ \text{values:} & v \in \mathcal{V} = \mathcal{L} \cup \{\mathit{null}\} & \text{heaps:} & h, k \in \mathcal{L} \rightharpoonup \mathcal{O} \\ \text{objects:} & \mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) & & \end{array}$$

The semantics of FJEU is defined as a standard big-step relation $(s, h) \vdash e \Downarrow v, h'$, which means that an FJEU expression $e$ evaluates in store $s$ and heap $h$ to the value $v$ and modifies the heap to $h'$. Due to reasons of limited space, figure 1 only shows the defining rules for some of the syntactic forms. A premise involving a partial function, like $s(x) = l$, implies the side condition $x \in \mathit{dom}(s)$.

## 2.4 Class Tables

A class table $\mathfrak{C}_0 = (A_0, M_0)$ models FJ's standard type system, where types are simply classes. The *field typing* $A_0 : (\mathcal{C} \times \mathcal{F}) \rightharpoonup \mathcal{C}$ assigns to each class $C$ and each field $f \in \mathit{fields}(C)$ the class of the field. The field class is required to be invariant with respect to subclasses of $C$. The *method typing* $M_0 : (\mathcal{C} \times \mathcal{M}) \rightharpoonup \overline{\mathcal{C}} \times \mathcal{C}$ assigns to each class $C$ and each method $m \in \mathit{methods}(C)$ a *method type*, which specifies the classes of the formal argument variables and of the result value. It is required to be contravariant in the argument classes and covariant in the result class with respect to subclasses of $C$.

$$\frac{l \notin dom(h) \qquad F = [f \mapsto null]_{f \in fields(C)}}{(s,h) \vdash \texttt{new}\ C \Downarrow l, h[l \mapsto (C,F)]} \qquad \frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2}$$

$$\frac{s(x) = l \qquad h(l) = (C,F) \qquad h' = h[l \mapsto (C, F[f \mapsto s(y)])]}{(s,h) \vdash x.f := y \Downarrow s(y), h'} \qquad \frac{s(x) = l \qquad h(l) = (C, \_) \qquad |\overline{x^m}| = |\overline{y}| = n \qquad s' = [this \mapsto s(x)] \cup [x_i^m \mapsto s(y_i)]_{i \in \{1,\ldots,n\}} \qquad (s',h) \vdash mtable(C,m) \Downarrow v, h'}{(s,h) \vdash x.m(\overline{y}) \Downarrow v, h'}$$

**Fig. 1.** Operational semantics of FJEU (extract)

## 3 Region Type System

In this section we define the base region type system, which serves as a main unifying calculus for pointer analysis and is given an interpretation and soundness proof. We assume an infinite set $\mathcal{R}$ of *regions* $r$, which are abstract memory locations. Each region stands for zero or more concrete locations. Different regions represent disjoint sets of concrete locations, hence they partition or *color* the memory. Two pointers to different regions can therefore never alias.

### 3.1 Refined Types and Subtyping

The region type system is a refinement of the plain type system: we equip classes with (possibly infinite) subsets from $\mathcal{R}$. For example, a location $l$ is typed with the *refined type* $C_{\{r,s\}}$ if it points to an object of class $C$ (or a subclass of $C$), and if $l$ is abstracted to either $r$ or $s$, but no other region. The *null* value can be given any type, while the type of locations must have a non-empty region set. The following table summarizes the definitions:

|  |  |
|---|---|
| Regions: | $r, s, t \in \mathcal{R}$ |
| Region sets: | $R, S, T \in \mathcal{P}(\mathcal{R})$ |
| Refined types: | $\sigma, \tau \in \mathcal{T} = \mathcal{C} \times \mathcal{P}(\mathcal{R})$ |

In the following, we use the notation $C_R$ instead of $(C, R)$ for types. Though the region identifier $s$ is already used for variable stores, the difference should be clear from the context. Since region sets are an over-approximation of the possible locations where an object resides, we can easily define a subtyping relation $<:$ based on set inclusion:

$$C_R <: D_S \iff R \subseteq S \ \wedge\ C \preceq D$$

We also extend subtyping to method types:

$$\overline{\sigma} <: \overline{\tau} \iff |\overline{\sigma}| = |\overline{\tau}| \ \wedge\ \forall i \in 1, \ldots, |\overline{\sigma}|.\ \sigma_i <: \tau_i$$
$$(\overline{\sigma}, \tau) <: (\overline{\sigma'}, \tau') \iff \overline{\sigma'} <: \overline{\sigma} \ \wedge\ \tau <: \tau'$$

## 3.2 Annotated Class Tables

We extend (plain) class tables $\mathfrak{C}_0$ to *annotated class tables* $\mathfrak{C} = (A^{get}, A^{set}, M)$.

- The *annotated field typings* $A^{get}, A^{set} : (\mathcal{C} \times \mathcal{R} \times \mathcal{F}) \rightharpoonup \mathcal{T}$ assign to each class $C$, region $r$ and field $f \in \mathit{fields}(C)$ the refined type of the field for all objects of class $C$ in region $r$. The field type is split into a covariant get-type $A^{get}$ for the data read from the field, and a contravariant set-type $A^{set}$ that is needed for data to be written to the field. This technique improves precision and is borrowed from Hofmann and Jost [20]. More formally, annotated field typings are *well-formed* if for all classes $C$, subclasses $D \preceq C$, regions $r$ and fields $f \in \mathit{fields}(C)$,
  - $A^{set}(C, r, f) <: A^{get}(C, r, f)$, and
  - $A^{get}(D, r, f) <: A^{get}(C, r, f)$ and $A^{set}(C, r, f) <: A^{set}(D, r, f)$.

  Also, the class component of $A^{get}(C, r, f)$ and $A^{set}(C, r, f)$ must be $A_0(C, f)$, i.e. invariant.
- The *annotated method typing* $M : (\mathcal{C} \times \mathcal{R} \times \mathcal{M}) \rightharpoonup \mathcal{P}(\overline{\mathcal{T}} \times \mathcal{T})$ assigns to each class $C$, region $r$ and method $m \in \mathit{methods}(C)$ an unbounded number of refined method types for objects of class $C$ in region $r$, enabling infinite polymorphic method types. This makes it possible to use a different type at different invocation sites (program points) of the same method. Even more importantly, the same invocation site can be checked in different type derivations with different method types. For every *well-formed* annotated method type, there must be an improved method type in each subclass: for all classes $C$, subclasses $D \preceq C$, regions $r$, and methods $m \in \mathit{methods}(C)$, we require
  - $\forall(\overline{\sigma}, \tau) \in M(C, r, m).\ \exists(\overline{\sigma'}, \tau') \in M(D, r, m).\ (\overline{\sigma}', \tau') <: (\overline{\sigma}, \tau)$.

  Again, the class components of the refined types $M(C, r, m)$ have to match the classes of the underlying unannotated method type $M_0(C, m)$.

In the following, we assume a fixed annotated class table $\mathfrak{C}$ with well-formed field and method typings.

## 3.3 Region Type System

The type system (see figure 2) derives judgements $\Gamma \vdash e : \tau$, meaning FJEU expression $e$ has type $\tau$ with respect to a *variable context* (store typing) $\Gamma : \mathcal{X} \rightharpoonup \mathcal{T}$ that maps variables to types.

The rule T-SUB is used to obtain weaker types for the expression. The rules T-LET, T-VAR, and T-IFINST are standard. The rule T-IFEQ exploits the fact that the two variables must point to the same object (or be *null*) in the `then` branch, therefore the intersection of the region sets can be assumed. In T-NULL, the *null* value may have any type (any class and any region set). In the rule T-NEW, we may choose any region $r$, which is an abstract location that includes (possibly among others) the concrete location of the object allocated by this expression.

$$\textsc{T-Sub}\ \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau} \qquad \textsc{T-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\textsc{T-Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{T-Null}\ \frac{}{\Gamma \vdash \texttt{null} : \tau}$$

$$\textsc{T-IfInst}\ \frac{x \in dom(\Gamma) \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\textsc{T-IfEq}\ \frac{\Gamma, x : C_{R \cap S}, y : D_{R \cap S} \vdash e_1 : \tau \qquad \Gamma, x : C_R, y : D_S \vdash e_2 : \tau}{\Gamma, x : C_R, y : D_S \vdash \texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\textsc{T-New}\ \frac{}{\Gamma \vdash \texttt{new}\ C : C_{\{r\}}}$$

$$\textsc{T-Invoke}\ \frac{\forall r \in R.\ \exists(\overline{\sigma'}, \tau') \in M(C, r, m).\ (\overline{\sigma'}, \tau') <: (\overline{\sigma}, \tau)}{\Gamma, x : C_R, \overline{y} : \overline{\sigma} \vdash x.m(\overline{y}) : \tau}$$

$$\textsc{T-GetF}\ \frac{\forall r \in R.\ A^{get}(C, r, f) <: \tau}{\Gamma, x : C_R \vdash x.f : \tau} \qquad \textsc{T-SetF}\ \frac{\forall r \in R.\ \tau <: A^{set}(C, r, f)}{\Gamma, x : C_R, y : \tau \vdash x.f := y : \tau}$$

**Fig. 2.** Region type system

When a field is read (T-GetF), we look up the type of the field in the $A^{get}$ table. As the variable $x$ may point to a number of regions, we need to ensure that $\tau$ is an upper bound of the get-types of $f$ over all $r \in R$. In contrast, when a field is written (T-SetF), the written value must have a subtype of the types allowed for that field by the $A^{set}$ table with respect to each possible region $r \in R$. Finally, the rule T-Invoke requires that for all regions $r \in R$ where the receiver object $x$ may reside, there must exist a method typing that is suitable for the argument and result types.

An FJEU program $P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ is *well-typed* if for all classes $C$, regions $r$, methods $m$ and method types $(\overline{\sigma}, \tau)$ such that $(\overline{\sigma}, \tau) \in M(C, r, m)$, the following judgement is derivable:

$$[\mathit{this} \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i \in \{1, \ldots, |\overline{x^m}|\}} \vdash \mathit{mtable}(C, m) : \tau$$

The polymorphic method types make the region type system very expressive in terms of possible analyses of a given program. Each method may have many types, each corresponding to a derivation of the respective typing judgment. In the different derivations, different regions may be chosen for new objects, and different types may be chosen for called methods. This flexibility provides the basis for our embedding of external pointer analyses. Moreover, since there may be infinitely many method types for each method, this system is equivalent to one which allows infinite unfoldings of method calls.

## 3.4 Interpretation

We now give a formal interpretation of the typing judgement in form of a soundness theorem. Afterwards, we continue by restricting the expressivity of the system and reformulating the rules in order to move towards an actual type checking algorithm. The underlying idea is that we only have to prove soundness once for the general system; for later systems, it suffices to show that they are special cases of the general system, such that the soundness theorem applies to these systems as well.

A *heap typing* $\Sigma, \Pi : \mathcal{L} \rightharpoonup (\mathcal{C} \times \mathcal{R})$ assigns to heap locations a static class (an upper bound of the actual class found at that location) and a region. Heap typings, a standard practice in type systems for languages with dynamic memory allocations [22], separate the well-typedness definitions of locations in stores and objects from the actual heap, thereby avoiding the need for a co-inductive definition for well-typed heaps in the presence of cyclic structures. Heap typings map locations to very specific types, namely those where the region set is a singleton. A heap typing thus partitions a heap into (disjoint) regions.

We define a typing judgment for values $\Sigma \vdash v : \tau$, which means that according to heap typing $\Sigma$, the value $v$ may be typed with $\tau$. In particular, the information in $\Sigma(l)$ specifies the type of $l$. Also, the typing judgment of locations is lifted to stores and variable contexts.

$$\frac{}{\Sigma \vdash \texttt{null} : \tau} \qquad \frac{\Sigma(l) = (C, r)}{\Sigma \vdash l : C_{\{r\}}} \qquad \frac{\Sigma \vdash v : \sigma \qquad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

$$\Sigma \vdash s : \Gamma \iff \forall x \in dom(\Gamma).\ \Sigma \vdash s(x) : \Gamma(x)$$

A heap $h$ is *well-typed* with respect to a heap typing $\Sigma$ and implicitly a field typing $A^{get}$, written $h \models \Sigma$, if the type for all locations given by $\Sigma$ are actually "valid" with respect to the classes of the objects, and if the field values are well-typed with respect to $A^{get}$ and $\Sigma$:

$$h \models \Sigma \iff \forall l \in dom(\Sigma).\ l \in dom(h)\ \wedge\ \Sigma \models h(l) : \Sigma(l)$$

where

$$\begin{aligned}\Sigma \models (C, F) : (D, r) \iff C \preceq D\ \wedge\ dom(F) = \mathit{fields}(C)\ \wedge \\ \forall f \in \mathit{fields}(C).\ \Sigma \vdash F(f) : A^{get}(C, r, f)\end{aligned}$$

As the memory locations are determined at runtime, the heap typings cannot be derived statically. Instead, our interpretation of the typing judgement $\Gamma \vdash e : \tau$ states that whenever a well-typed program is executed on a heap that is well-typed with respect to some typing $\Sigma$, then the final heap after the execution is well-typed with respect to some possibly larger heap typing $\Pi$. The typing $\Pi$ may be larger to account for new objects that may have been allocated during execution, but the type of locations that already existed in $\Sigma$ may not change. More formally, a heap typing $\Pi$ *extends* a heap typing $\Sigma$, written $\Pi \sqsupseteq \Sigma$, if $dom(\Sigma) \subseteq dom(\Pi)$ and $\forall l \in dom(\Sigma).\ \Sigma(l) = \Pi(l)$.

**Theorem 1 (Soundness Theorem).** *Fix a well-typed program $P$. For all $\Sigma, \Gamma, \tau, s, h, e, v, k$ with*

$$\Gamma \vdash e : \tau \quad \textit{and} \quad \Sigma \vdash s : \Gamma \quad \textit{and} \quad (s,h) \vdash e \Downarrow v, k \quad \textit{and} \quad h \models \Sigma$$

*there exists some $\Pi \sqsupseteq \Sigma$ such that*

$$\Pi \vdash v : \tau \quad \textit{and} \quad k \models \Pi.$$

*Proof.* By induction over the derivations of the operational semantics and the typing judgement. Details can be found on the authors' homepage [21]. We have also developed a formalization in Isabelle/HOL.

## 4 Parametrized Region Type System

Our next goal is to use the type system for the automatic algorithmic verification of results of external analyses. In this section, we focus on the first step in this direction by showing how to *interpret* given analysis results in the type system. We then outline how to implement an algorithmic type checking algorithm for the automatic verification.

The interpretation of given results requires the reformulation of the above type system to explicitly model abstraction principles that are fundamental for a number of different pointer analysis techniques [7]. We focus on two general classes of abstractions: the abstraction of the call graph using contexts, and the abstraction of objects on the heap. The region type system is equipped with parameters that can be instantiated to specific abstraction principles. We show that the parametrized version of the type system arises as a specialization of the general region type system.

In the following, the notion of program points is made explicit by annotating expressions with *expression labels* $i \in \mathcal{I}$: we write $[e]^i$ for FJEU expressions, where $e$ is defined as before. An FJEU program is well-formed if each expression label $i$ appears at most once in it. In the following, we only consider well-formed programs, and simply write $e$ instead $[e]^i$ if the expression label $i$ is not important.

### 4.1 Abstraction Principles

Context-insensitive pointer analyses examine each method exactly once. Their result provides for each method a single pointer structure specification which is used for every call of that method. *Context-sensitive* algorithms improve precision: each method may be analyzed multiple times under different *contexts*, so that different specifications can be used for different calls to the same method. A context-sensitive algorithm settles on a specific (finite) set of contexts, and produces for each method one pointer structure specification per context. Pointer structure specifications correspond to our method types. We therefore model the concept of contexts by introducing a finite abstract set of contexts $\mathcal{Z}$, and by parametrizing the method typing function $M$ to associate one method type per context in $\mathcal{Z}$.

The choice of contexts and specifications for each method call depends on the analysis in question. For example, call-site sensitive algorithms [23] process each method once for each program point where the method is called. Receiver-object sensitive analyses [24] differentiate pointer structure specifications of a method according to the abstraction of the invoking object. More powerful analyses use *call stacks* as contexts to differentiate method calls. For example, a method $m$ may be analyzed for each possible call-site stack that may occur at the time when $m$ is called. Similarly, receiver-object stacks can be used. In other words, one considers the *call graph* of the program. A method $m$ is then represented by a node in this graph, and a context corresponds to a possible path that leads to the node. As there may be infinitely many paths in recursive programs, the number of paths needs to be restricted by some mechanism. A common way is to consider only the last $k$ entries on the stack ($k$-CFA [10]), or to collapse each strongly connected component into a node, thereby eliminating recursive cycles from the set of possible paths [1,25]. Following this observation, we employ a general *context transfer function* $\phi$ which represents the edges in the abstract call graph. The function selects a context for the callee based on the caller's context, the class of the receiver object, its region, the method name, and the call site.

Another abstraction principle is the object abstraction, i.e. the abstract location assigned to allocated objects. This corresponds to our concepts of regions. As pointer analysis algorithms differentiate only finitely many abstract objects, we can restrict the set of regions $\mathcal{R}$ to a finite size.

A common abstraction is to distinguish objects according to their allocation site and/or their class. More precise analyses also take into account the context under which allocation takes place. For example, in object-sensitive analysis by Milanova et al. [24], objects are distinguished by their own allocation site and the allocation site of the `this` object. Objects may also be distinguished by the call site of the invoked method, a technique called *heap specialization* [26]. We model these concepts by an *object abstraction function* $\psi$ that assigns the region for the new object, given the allocation site and the current method context. Our system is by design class-sensitive, as the class information is part of the type.

Figure 3 summarizes the four parameters of our system: contexts, context transfer function, regions, and object abstraction function. Also, it shows how to instantiate the parameters to obtain various standard abstraction principles.

The parametrized method typing $\hat{M} : (\mathcal{C} \times \mathcal{R} \times \mathcal{Z} \times \mathcal{M}) \rightharpoonup \overline{\mathcal{T}} \times \mathcal{T}$ replaces the annotated polymorphic method typing $M$ in the annotated class table. It is well-formed if for all classes $C$ and subclasses $D \preceq C$, regions $r \in \mathcal{R}$, methods $m \in \mathit{methods}(C)$, and contexts $z \in \mathcal{Z}$, it holds $\hat{M}(D, r, z, m) <: \hat{M}(C, r, z, m)$.

$$\begin{aligned} \text{Regions (finite): } & r,s,t \in \mathcal{R} \\ \text{Contexts (finite): } & z \in \mathcal{Z} \\ \text{Context transfer function: } & \phi \in \mathcal{Z} \times \mathcal{C} \times \mathcal{R} \times \mathcal{M} \times \mathcal{I} \to \mathcal{Z} \\ \text{Object abstraction function: } & \psi \in \mathcal{Z} \times \mathcal{I} \to \mathcal{R} \end{aligned}$$

| *set of regions* | *object abstraction function* | *principle* |
|---|---|---|
| $\mathcal{R} = \mathcal{I}$ | $\psi(z,i) = i$ | allocation site abstraction |
| $\mathcal{R} = \mathcal{Z} = \mathcal{I} \times \mathcal{I}$ | $\psi((i_1,i_2),i_0) = (i_0,i_1)$ | object-sensitive allocation site abstraction |
| $\mathcal{R} = \mathcal{Z} = \mathcal{I}$ | $\psi(i_c,i) = i_c$ | heap specialization |

| *set of contexts* | *context transfer function* | *principle* |
|---|---|---|
| $\mathcal{Z} = \{z_0\}$ | $\phi(z,C,r,m,i) = z_0$ | context-insensitivity |
| $\mathcal{Z} = \mathcal{R}$ | $\phi(z,C,r,m,i) = r$ | object-sensitive 1-CFA |
| $\mathcal{Z} = \bigcup_{n\in\{1,\dots,k\}} \mathcal{M}^n$ | $\phi(z,C,r,m,i) = (m :: z)\|_k$ | method identifier $k$-CFA |
| $\mathcal{Z} = \{\overline{i} \in \sigma(\mathcal{I}') \mid \mathcal{I}' \subseteq \mathcal{I}\}$ | $\phi(z_1,C,r,m,i) = z_2$ s.th. $IE_c(z_1,i,z_2,m)$ | CFA with eliminated recursive cycles |

where

- $\sigma(X)$ is the set of all permutations of $X$
- $L|_k$ is the truncation of list $L$ after the first $k$ elements
- $IE_c$ is the call graph relation of [25] where recursive cycles have been replaced by single nodes

**Fig. 3.** The four type system parameters, and possible instantiations to common abstraction principles

## 4.2 The Parametrized Type System

The parametrized type system extends the typing judgment of the general region type system by a context component $z$. With the exception of the following two rules, all rules remain as presented in section 3 (with the addition of the context $z$ in each judgement):

$$\text{TP-New}\ \frac{r = \psi(z,i)}{\Gamma\ ;\ z\ \vdash\ [\texttt{new}\ C]^i\ :\ C_{\{r\}}}$$

$$\text{TP-Invoke}\ \frac{\forall r \in R.\ \hat{M}(C,r,\phi(z,C,r,m,i),m) <: (\overline{\sigma},\tau)}{\Gamma, x : C_R, \overline{y} : \overline{\sigma}\ ;\ z\ \vdash\ [x.m(\overline{y})]^i\ :\ \tau}$$

While in the previous system any region $r$ could be chosen for new objects, we have restricted this flexibility in the TP-New rule to the region specified by $\psi$. Moreover, instead of allowing arbitrarily many types per method, from which any type could be selected for invocations, we now have one type determined by the context $z$ that is selected by the $\phi$ function in the TP-Invoke rule.

For a given parametrized method typing $\hat{M}$, we define a the corresponding polymorphic method typing $M(C, r, m) := \bigcup_{z \in \mathcal{Z}} \{\hat{M}(C, r, z, m)\}$. The rules of the parametrized system are derivable in the previous system: The only changed rules TP-New and TP-Invoke have more restrictive premises than their counterparts T-New and T-Invoke. Hence if $\Gamma\ ;\ z\ \vdash\ e\ :\ \tau$ can be derived from some $\hat{M}$ in the parametrized system, then $\Gamma\ \vdash\ e\ :\ \tau$ can be derived in the previous system with respect to the corresponding method typing $M$.

A method table is *well-typed* for $\hat{M}$ if for all classes $C$, contexts $z$, regions $r$, and methods $m$ such that $\hat{M}(C, r, z, m) = (\overline{\sigma}, \tau)$, the judgement $\Gamma\ ;\ z\ \vdash$ $\mathit{mtable}(C, m)\ :\ \tau$ can be derived with $\Gamma = [\mathit{this} \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i \in \{1,\ldots,|\overline{x^m}|\}}$. It is easy to see that if a method table is well-typed with respect to $\hat{M}$ in the parametrized system, then it is also well-typed with respect to the corresponding method typing $M$ in the general system. Therefore, the soundness theorem is applicable to the parametrized region type system.

### 4.3 Algorithmic Type Checking

The parametrized type system can be rewritten into a syntax directed form, from which one can directly read off an algorithm $A(\Gamma, e) = \tau$ that computes "from left to right" the type $\tau$ of an expression $e$ based on a store typing $\Gamma$. For this, we eliminate the subtyping rule, and instead specify the most precise resulting type $\tau$ for each expression, similarly to the approach taken by Pierce [22]. The soundness proof shows that the internalisation of the subtyping rule is correct, i.e. that the judgements derived with the algorithmic type system can also be derived with the parametrized type system. To demonstrate the verification capabilities of the algorithmic type system, we have also developed a context-sensitive pointer analysis algorithm for FJEU, described declaratively as a set of recursive Datalog rules in the style of Whaley and Lam [25]. The full algorithmic type system, its soundness proof and the sample pointer analysis algorithm can be found on the authors' homepage [21].

## 5 String Analysis

String analysis is a dataflow analysis technique to determine possible string values (character sequences) that may occur during the execution of a program. Since strings appear as objects in Java, it is natural to implement a string analysis by building on pointer analysis: string objects are identified and tracked by the pointer analysis, while their possible values are determined by the string analysis.

We now equip the FJEU language with special string objects and operations to give a simplified formalization of Java's `String` class, and extend the region type system to enable the verification of pointer analyses of string objects. Afterwards, we show how to use the region information to interpret the results of a specific string analysis.

## 5.1 FJEU with Strings

The language FJEUS is an extension of FJEU with operations to create and concatenate strings. While the `String` class in Java is just another class in the Java class hierarchy, it is regarded as a separate type in FJEUS. This allows us to treat string objects differently: An object of class *String* in FJEUS is simply a string value (character sequence) on the heap. The meta-variable $w$ ranges over character sequences $\mathcal{W}$, and $W$ ranges over sets of string values. We rely on a given sequence concatenation function $+$.

$$\begin{array}{rrcl} \text{character sequences:} & w & \in & \mathcal{W} \\ \text{sets of character sequences:} & W & \in & \mathcal{P}(\mathcal{W}) \\ \text{extended expressions:} & \mathcal{E} \ni e & ::= & \ldots \mid \texttt{new String}(w) \mid x.\mathit{concat}(y) \end{array}$$

$$\begin{array}{rrcl} \text{character sequence concatentation:} & + & \in & \mathcal{W} \times \mathcal{W} \to \mathcal{W} \\ \text{heaps:} & h, k & \in & \mathcal{L} \rightharpoonup \mathcal{O} \cup \mathcal{W} \end{array}$$

The expression $\texttt{new String}(w)$ allocates a new string object with the character sequence $w$ on the heap. The string operation $x.\mathit{concat}(y)$ has its own special semantics and is implemented with the $+$ operator. As we only model strings with non-mutable values in the language, a string concatenation always creates a new string object on the heap. The operational semantics is extended as follows:

$$\frac{l \notin \mathit{dom}(h)}{(s, h) \vdash \texttt{new String}(w) \Downarrow l, h[l \mapsto w]}$$

$$\frac{s(x) = l_1 \qquad s(y) = l_2 \qquad h(l_1) = w_1 \qquad h(l_2) = w_2 \\ l \notin \mathit{dom}(h) \qquad w = w_1 + w_2}{(s, h) \vdash x.\mathit{concat}(y) \Downarrow l, h[l \mapsto w]}$$

Note that this rather modest extension is intended to keep the formalization simple. Other extensions could include more string operations, or mutable string objects that model Java's `StringBuffer` class.

## 5.2 Pointer Analysis for String Objects

We extend the region type system from section 3 to accommodate the new string objects. We distinguish references to "proper" objects and to string objects: a type is either a class with a region set ($C_R$), or the special `String` class with a region set ($\texttt{String}_R$). The `String` class is independent from other classes in the class hierarchy.

$$\text{types: } \sigma, \tau \in \mathcal{T} = (\mathcal{C} \cup \{\texttt{String}\}) \times \mathcal{P}(\mathcal{R})$$

$$\frac{C \preceq D \qquad R \subseteq S}{C_R <: D_S} \qquad\qquad \frac{R \subseteq S}{\texttt{String}_R <: \texttt{String}_S}$$

Fields and method typings may now include the $\texttt{String}_R$ type. Also, all existing typing rules from section 3 remain unchanged. In particular, the *null* value may be assigned a `String` type. A field $x.f$ may only be accessed if $x$ is a (non-`String`) class $C$; similarly for method calls $x.m$. These are the two additional typing rules for `new String`$(str)$ and $x.concat(y)$:

$$\frac{}{\Gamma \ \vdash \ \texttt{new}\,\texttt{String}(w) \ : \ \texttt{String}_{\{r\}}}$$

$$\frac{}{\Gamma, x : \texttt{String}_R, y : \texttt{String}_S \ \vdash \ x.\mathit{concat}(y) \ : \ \texttt{String}_{\{t\}}}$$

The heap typings $\Sigma, \Pi : \mathcal{L} \rightharpoonup (\mathcal{C} \cup \{\texttt{String}\}) \times \mathcal{R}$ may now also map locations to the `String` class. We extend the well-typed value relation $\Sigma \vdash v : \tau$ accordingly:

$$\frac{}{\Sigma \vdash \texttt{null} : \tau} \qquad \frac{\Sigma(l) = (C, r)}{\Sigma \vdash l : C_{\{r\}}} \qquad \frac{\Sigma(l) = (\texttt{String}, r)}{\Sigma \vdash l : \texttt{String}_{\{r\}}}$$

$$\frac{\Sigma \vdash v : \sigma \qquad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

The definition of well-typed heaps additionally requires well-typed character sequences (last line):

$$\begin{aligned} h \models \Sigma &\iff \forall l \in \mathit{dom}(\Sigma).\ l \in \mathit{dom}(h) \ \wedge \ \Sigma \models h(l) : \Sigma(l) \\ \Sigma \models (C, F) : (D, r) &\iff C \preceq D \ \wedge \ \ldots \ \text{(as before)} \\ \Sigma \models w : (\texttt{String}, r) &\iff \mathsf{PROP}(w, r) \end{aligned}$$

In other words, the property that a heap $h$ is well-typed with respect to $\Sigma$ now includes the condition that for all locations $l$ such that $\Sigma(l) = (\texttt{String}, r)$, $h(l)$ contains a string value $w$ that satisfies a certain property $\mathsf{PROP}$ with respect to $r$. For the moment, assume $\mathsf{PROP}$ is simply *True*. The proof of the soundness theorem is extended in a straight-forward way for the extensions described above. Moreover, the type system can be parametrized in the same fashion as described in section 4: as both string operations create new objects, we use the $\psi$ function to determine the region of these objects.

In the following subsection, we present an analysis that can help to prevent cross-site scripting attacks, and give a semantic formalization by instantiating the string property $\mathsf{PROP}$.

## 5.3 String Analysis with Operation Contexts

In a typical cross-site scripting scenario, a user input is embedded into a string that is executed or interpreted. To prevent the injection of malicious code, one wants to track how a string is constructed, and ensure that its executable parts originate from trusted sources, like string literals in the program code.

We therefore add a string operation context $\Omega$, which expresses the possible string operations that objects in specific regions may be the result of. The typing rules are extended with constraints on $\Omega$.

$$\begin{array}{ll}\text{string operations: } & \mathcal{O} \ni \omega ::= \texttt{newstring}\, w\ r \mid \texttt{concat}\ r\ s\ t \mid \texttt{unknown}\, W\ r \\ \text{operation context: } & \Omega \ \in\ \mathcal{P}(\mathcal{O})\end{array}$$

$$\frac{\texttt{newstring}\, w\ r \in \Omega}{\Gamma\ \vdash\ \texttt{new}\ \texttt{String}(w)\ :\ \texttt{String}_{\{r\}}}$$

$$\frac{\forall r \in R, s \in S.\ \texttt{concat}\ r\ s\ t \in \Omega}{\Gamma, x : \texttt{String}_R, y : \texttt{String}_S\ \vdash\ x.concat(y)\ :\ \texttt{String}_{\{t\}}}$$

Informally, an operation $\texttt{newstring}\, w\ r \in \Omega$ means that region $r$ may include a string $w$. $\texttt{concat}\ r\ s\ t$ means that region $t$ may include a string object that is obtained by concatenating some strings from regions $r$ and $s$. The operation $\texttt{unknown}\ W\ r$ means that strings in region $r$ may be from the set $W$. This is useful for external methods whose types are given to but not verified by the type system. Whenever more primitive string operations are added to the language, the set $\mathcal{O}$ may be extended accordingly.

Formally, we define a semantic interpretation $\Omega[\![r]\!]$ that gives the possible values for string objects in region $r$. It is defined as the smallest set satisfying the following conditions:

$$\begin{array}{rcl}\texttt{newstring}\, w\ r \in \Omega & \Rightarrow & w \in \Omega[\![r]\!] \\ \texttt{concat}\ r\ s\ t \in \Omega & \Rightarrow & \forall w_1 \in \Omega[\![r]\!], w_2 \in \Omega[\![s]\!].\ w_1 + w_2 \in \Omega[\![t]\!] \\ \texttt{unknown}\, W\ r \in \Omega & \Rightarrow & W \subseteq \Omega[\![r]\!]\end{array}$$

After instantiating the string property as $\mathsf{PROP}(w, r) \equiv w \in \Omega[\![r]\!]$, the relation $h \models \Sigma$ ensures that string values on the heap are indeed in the interpretation of the string operation context.

Apart from this extensional interpretation, the string operation context and the typing of external methods also contain intensional information about the origin and the possible constructions of strings in a specific region, which enables the verification of more complex string policies.

For example, consider the following string-manipulating program that relies on external functions *getUserInput()* to retrieve data from the user, *escape-HTML(s)* that quotes all HTML tags in string $s$ and returns the result as a new string, and *output(s)* that outputs the string $s$.

```
let firstPart = new String("<sometag>")
 in let contents = getUserInput()
     in let escContents = escapeHTML(contents)
         in output(firstPart.concat(sanContents))
```

The security policy is that the output may only be the result of a concatenation of a string literal with a string that does not contain HTML tags. The policy is expressed using the following types for external methods and the string operation context:

$$\begin{aligned} \mathit{getUserInput} &: \texttt{unit} \longrightarrow \texttt{String}_{\{q\}} & \Omega = \{\, &\texttt{newstring "<sometag>"}\ l, \\ \mathit{escapeHTML} &: \texttt{String}_{\{q\}} \longrightarrow \texttt{String}_{\{s\}} & &\texttt{unknown}\ \mathcal{W}\ q,\ \ \texttt{unknown}\ \hat{\mathcal{W}}\ s, \\ \mathit{output} &: \texttt{String}_{\{t\}} \longrightarrow \texttt{unit} & &\texttt{concat}\ s\ l\ t,\ \ \texttt{concat}\ l\ s\ t \quad \} \end{aligned}$$

($\hat{\mathcal{W}}$ is the set of all strings that do not contain HTML tags, and `unit` is a unit type, which could be modeled in FJEUS as a null type like $\texttt{String}_{\emptyset}$.) The external function *getUserInput* returns strings with arbitrary values of the set $\mathcal{W}$ in region $q$ ("questionable"), which are converted by *escapeHTML* into strings of region $s$ ("sanitized"), which are assumed to not contain any HTML tags (set $\hat{\mathcal{W}}$). For the literal `firstPart`, the type checker can assign the region $l$ ("literals"), as the literal value in $\Omega$ matches. The *output* function only accepts strings from region $t$ ("trusted"), which must be, according to $\Omega$, a concatenation of strings from region $l$ and (HTML tag-free) region $s$. Therefore, the typability of the program proves that the security policy is indeed fulfilled. The example demonstrates that handling trusted sanitizing functions is actually a strength of type-based presentations: simply assign an appropriate type to a function if you believe the associated semantic property of the function.

The approach is related to the work by Christensen et al. [8] on the analysis of string values using context-free grammars with operation productions. The symbolic string operations in the context correspond to nodes in their annotated flow graph, and their semantics of the flow graph resembles our interpretation of $\Omega[\![r]\!]$. Similarly, Crégut and Alvarado [9] have presented an algorithm that tracks string objects with pointer analysis, and collects intensional information about the string operations applied to them. We thus expect that aspects of the results of these algorithms are verifiable in our system. Our approach is also related to taint analysis [27], as the region identifiers can convey information about the trustworthiness of strings, which is preserved throughout assignments and method invocations.

## 6 Discussion

We presented a framework for classifying alias analyses for Java-like languages, given by a hierarchy of region-based type systems. We demonstrated how existing disciplines arise as instantiations of our framework and may be given a uniform interpretation by embedding their results in a single type system. We also gave an algorithmic variant of the type system, thus enabling syntax-directed type-checking and hence validation of analyses results. Finally, we showed how our framework may be extended to string analyses. In the following, we briefly discuss specific design decisions, and outline future work.

To our knowledge, most existing pointer analyses express their results at a coarse-grain level of syntactic structure such as methods. In accordance with this, we employed a phrase-based formulation of type systems and interpreted judgements with respect to a big-step evaluation semantics. An extension of the interpretation to include non-terminating executions appears possible, using techniques as in [28].

Our framework does not aim to be flow- or path-sensitive. We see these concepts as orthogonal to the central idea of our paper, namely interpreting context-sensitivity using polyvariant types. Nevertheless, we acknowledge the increasing relevance of flow and path sensitivity in recent work on pointer analysis. The T-IfEq rule illustrates a possible extension of the type system with path-sensitive capabilities: the information from the branching expression is used to refine the analysis for the "then" branch of the conditional. Moreover, subderivations of a judgement contain implicitly more fine-grained (non-)alias relationships applicable at intermediate program points, and include aspects of flow-sensitivity as any variable may be associated with different types in a derivation. Arguably, local alias assertions could be made more explicit by moving to a small-step operational regime and/or formulations of the type systems that are inspired by abstract interpretation and yield a global specification table with entries for all program points [29]. However, the use of evaluation-style judgements greatly simplifies soundness proofs at least at the level of methods, as recursive calls follow the type derivation structure.

The Doop framework [5] enables the definition of highly precise and efficient pointer analyses declaratively using Datalog rules. While the authors do not seem to aim at a fully formal correctness proof that interprets the Datalog rules and relations with respect to the semantics of the Java language, they take great care to separate the essential aspects of analysis techniques from implementation details of the algorithm. We intend to look at the ideas in their work in order to find ways to adapt our type system to more language features and pointer analyses.

In addition to type systems for pointer alias analysis, the literature contains the terminology "type-based alias analysis" [30]. The latter term appears to mean that datatype or static class information is used to *improve the precision* of an alias analysis, while region type systems directly *include* the points-to relations in the types. However, as our system extends the ordinary type system of Java, it arguably also encompasses type-based alias analyses.

Having been developed in order to embed static analysis results, it is not surprising that the type systems over-approximate their semantic guarantee. Thus, failure to validate the *legitimate* result of a specific analysis may be rooted in either an incorrect interpretation of the analysis into our framework or the fact that the analysis is more precise than the host type system. A particular direction in which our type system (and some analyses) might be generalized is shape analysis [31]. Another interesting recent development that our work does not accommodate is the analysis of programs that use reflection [27]; this would require a fundamental understanding of the semantic analysis of reflection.

Although we have only examined the *results* of pointer analysis algorithms, the algorithms can be seen as an external means of type inference. It seems promising to further investigate the *implementations* of these algorithms, and to recreate their logic in the type system in order to obtain a parametric type system with a fully automatic (internal) type inference. Alternatively, the identification of abstraction principles could also propose a way to parametrize existing pointer analysis implementations.

Regarding the string analysis, we have concentrated on the previously-noted observation that the precision of the analysis benefits from the availability of (non-)aliasing information [9]. In principle, the benefits may be mutual. For example, the method call $x.concat(y)$ on a `String` in Java actually returns the reference $x$ if $y$ has a length of zero. If the length of $y$ can be obtained from a string analysis, this information helps to improve the region set for the result type in the rule for concatenation. Mutual dependencies between aliasing and string analyses may thus be an interesting topic for future work.

Also, we have only outlined the use of a verified string analysis for security policies. In collaboration with SAP's Sophia-Antipolis-based research lab on security we plan to extend the type system with string operation contexts to verify the absence of cross-site scripting attacks in concrete scenarios.

*Acknowledgements.* We would like to thank Pierre Crégut for making us aware of the interplay between pointer and string analyses and providing us with several pointers to the literature, as well as Keqin Li (SAP research) for giving us insights to concrete cross-site scripting scenarios and the corresponding string policies. This work was supported by the Information Society Technologies programme of the European Commision, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

## References

1. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: 1994 Conference on Programming Language Design and Implementation (PLDI 1994), pp. 242–256. ACM, New York (1994)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
3. Steensgaard, B.: Points-to analysis in almost linear time. In: 23rd Symposium on Principles of Programming Languages (POPL 1996), pp. 32–41. ACM, New York (1996)
4. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: 2003 Conference on Programming Language Design and Implementation (PLDI 2003). ACM, New York (2003)
5. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009), pp. 243–262. ACM, New York (2009)
6. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: 1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999). ACM, New York (1999)

7. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694. Springer, Heidelberg (2003)
9. Crégut, P., Alvarado, C.: Improving the Security of Downloadable Java Applications With Static Analysis. Electronic Notes in Theoretical Computer Science 141(1), 129–144 (2005)
10. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU-CS-91-145 (1991)
11. Banerjee, A., Jensen, T.: Modular control-flow analysis with rank 2 intersection types. Mathematical Structures in Computer Science 13(1), 87–124 (2003)
12. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. SIGPLAN Not. 44(1), 226–238 (2009)
13. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: 15th Symposium on Principles of Programming Languages (POPL 1988), pp. 47–57. ACM, New York (1988)
14. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: Principles and Practice of Decl. Prog. (PPDP 2007). ACM, New York (2007)
15. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
16. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: 26th Symposium on Principles of Programming Languages (POPL 1999), pp. 262–275. ACM, New York (1999)
17. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: 2002 Conference on Programming Language Design and Implementation (PLDI 2002), pp. 1–12. ACM, New York (2002)
18. Lhoták, O.: Program Analysis using Binary Decision Diagrams. PhD thesis, McGill University (January 2006)
19. Lenherr, T.: Taxonomy and Applications of Alias Analysis. Master's thesis, ETH Zürich (2008)
20. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
21. Beringer, L., Grabowski, R., Hofmann, M.: Verifying Pointer and String Analyses with Region Type Systems: Soundness proofs and other material (2010), `http://www.tcs.ifi.lmu.de/ grabow/regions`
22. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
23. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications. Prentice Hall International, Englewood Cliffs (1981)
24. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. 14(1), 1–41 (2005)
25. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. SIGPLAN Not. 39(6), 131–144 (2004)
26. Nystrom, E.M., Kim, H.S., Hwu, W.W.: Importance of heap specialization in pointer analysis. In: 5th Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004). ACM, New York (2004)

27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: 2009 Conference on Programming Language Design and Implementation (PLDI 2009), pp. 87–97. ACM, New York (2009)
28. Beringer, L., Hofmann, M., Pavlova, M.: Certification Using the Mobius Base Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 25–51. Springer, Heidelberg (2008)
29. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
30. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: 1998 Conference on Programming Language Design and Implementation (PLDI 1998), pp. 106–117. ACM, New York (1998)
31. Loncaric, S.: A survey of shape analysis techniques. Pattern Recognition 31, 983–1001 (1998)