Type-Based Enforcement of Secure Programming Guidelines — Code Injection Prevention at SAP

[Extended Version]

Robert Grabowski[1], Martin Hofmann[1], and Keqin Li[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität,
Oettingenstrasse 67, D-80538 München, Germany
{robert.grabowski,martin.hofmann}@ifi.lmu.de
[2] SAP Research, France
805 Avenue du Dr Maurice Donat; 06254 Mougins Cedex; France
keqin.li@sap.com

Abstract. Code injection and cross-site scripting belong to the most common security vulnerabilities in modern software, usually caused by incorrect string processing. These exploits are often addressed by formulating programming guidelines or “best practices”.
In this paper, we study the concrete example of a guideline used at SAP for the handling of untrusted, potentially executable strings that are embedded in the output of a Java servlet. To verify adherence to the guideline, we present a type system for a Java-like language that is extended with refined string types, output effects, and polymorphic method types.
The practical suitability of the system is demonstrated by an implementation of a corresponding string type verifier and context-sensitive inference for real Java programs.

1 Introduction

Modern software typically must be able to interact with the whole world. For example, almost all business software provides access via a web interface, thereby exposing it to worldwide security threats. At the same time, one can no longer rely on the high skill and experience of specialist programmers.

To address this issue, programming guidelines and “best practices” have been developed [1] that summarize and condense the expert knowledge and make it available to a larger community. The extent to which such guidelines are correctly applied, however, is left to the responsibility of the programmer. It is thus desirable to use automatic methods to check that programming guidelines have been correctly and reasonably applied without compromising the flexibility of writing code. We propose to use a type-based approach for this purpose.

As a proof-of-concept, in this paper, we concentrate on a guideline used at SAP [2, 3] to counter the particular security threat posed by *code injection*, where

a malicious user inputs strings containing code fragments that may potentially be executed — assuming there exists a corresponding vulnerability on the server side. This scenario is surprisingly common, and indeed in the top ten list of most critical web application security risks published by the Open Web Application Security Project (OWASP), the top two positions are related to code injection [4].

1.1 Code Injection

For a simple example, consider a wiki web service that allows the creation of a new page by sending arbitrary text `contents` to the server, which is then displayed as a HTML page to a visitor using the following code on the server:

```
output("<body>" + contents + "</body>");
```

If `contents` is the malicious string

```
<script src="http://attacker.com/evil.js" />
```

then loading the generated HTML page will automatically execute a script from a different server. There are of course numerous more sophisticated attacks [3, 5, 6], for example such that an attacker can spy on authentication cookies.

All these attacks share a common pattern: they usually arise whenever untrusted input, typically a string, is combined to form a piece of executable code, e.g. an SQL query, an HTML page, or a script. The vulnerability is caused by part of the user input not being processed in the intended way. The attack can be countered by preprocessing the input prior to concatenation with code fragments. In our example, the preprocessed ("sanitized") input could be

```
&lt;script src="http://attacker.com/evil.js" /&gt;
```

In general, there are different forms of code injection, the most popular being cross-site scripting (XSS), SQL injection, XPath injection, and dynamic evaluation. In this work, we will focus on XSS attacks like the one presented above, though the results could be applied to similar code injection attacks as well.

1.2 Programming Guidelines

A number of program analysis techniques have been proposed that directly address code injection and related attacks [7–10, 5]. In our view, it is usually hard to specify code injection attacks exactly and to define what programs are subject to those attacks. As a result, such tools are useful in bug hunting, but may lack a rigorously specified and predictable behaviour.

In contrast, a programming guideline or practice can be formalized exactly, thus providing the semantic foundation for a sound procedure that can be shown to enforce the guideline. While strict compliance does not necessarily prevent all attacks, we argue that by separating the somewhat imprecise task of preventing an attack from the strict enforcement of a policy, the overall security can be improved, and the entire analysis can be simplified. As a side effect, programming guidelines help to prevent attacks already at the design time of the program. A

comprehensive archive of programming guidelines used in industry is maintained by the OWASP Application Security Verification Standard Project [1].

Although one might argue that programming guidelines constitute just a special class of security properties, they are the easiest to handle in software development and quality assurance, and possibly also when it comes to legal aspects. Despite their ubiquity and importance in practice, such guidelines have rarely been an application target in works on formal sound program analysis.

In the end, the focus on safe coding practices entails a more fundamental shift in the overall security model: Instead of assuming an adversary with varying attack capabilities, we primarily target a well-intentioned, generally trustworthy programmer who occasionally makes mistakes that enable these attacks.

Enforcing a guideline with automatic methods always constitutes a delicate navigation between efficiency, accuracy, and intuitiveness. Typically, programming guidelines, while focusing on syntax, carry some semantic component and therefore are in general undecidable. Now, if the automatic method raises too many false alarms then programmers will ignore the results of the method. The tool should run efficiently to be useful during coding. Finally, the method should be predictable, i.e., there should be a well-described approximation of the guideline in question which is then decided accurately by the automatic method.

In this work we focus on a particular programming guideline used at SAP to counter code injection attacks [2, 3]. Basically, this guideline requires the use of "sanitizing" functions that quote or escape characters that could otherwise cause the interpretation of parts of the strings as executable code. There is, however, no single sanitizing function that should be used for arbitrary user input; rather must one out of four such functions be selected according to the string context into which the user input is to be embedded. This makes static enforcement of the guideline a nontrivial task because we need to explore the possible string contexts as accurately as possible merely by analysing the program text.

We argue that a type system is an appropriate analysis technique, as it is a syntactic framework that classifies phrases according to categories such as the values they compute, or their origin. In addition to the analysis per se, this paper practically demonstrates that a type-based analysis provides a necessary degree of accuracy in this scenario.

1.3 Benefits of Type Systems

Using types as the basis of the analysis has several advantages. Programming guidelines are meant to be understood by the programmer, hence it is natural to enforce them using a technique that builds on the familiar Java type system.

From a theoretical point of view, it has been noticed [11] that type systems can be used to draw a clear distinction between the declarative statement of a program property by means of a typability relation, and its automatic verification using a type inference algorithm. The declarative definition of valid typing judgements simplifies the formulation of a rigorous soundness proof [12]. Also, type derivations can act as proof certificates for the desired program property.

Type systems have been successfully used not only to prove data type safety, but also to enforce security requirements such as noninterference properties used in information flow security. To our knowledge, however, type systems have not yet been used specifically to implement *programming guidelines* to prevent code injection vulnerabilities.

1.4 Contributions

The goal of this paper is to provide a type system that ensures that a Java programmer follows a given programming guideline to prevent code injection attacks. The main contributions are:

1. the identification of a new subfield: using type systems for the automatic enforcement of programming guidelines;
2. the development of an expressive type system for a particular programming guideline used at SAP for the prevention of code injection, this includes the formalization of the guideline with finite state machines;
3. the development of an accompanying implementation;
4. enhancing the accuracy of type-based string analysis to come close to that of black-box analyses without certification.

Note that for a strict formalization of the security property, the type system is defined on a theoretic core language "FJEUS" in the style of Featherweight Java. The implementation, however, works on actual Java source code.

We proceed as follows: we show how a particular class of XSS programming guidelines can be formalized as a finite state machine (Section 2). In Section 3, the core language is defined, followed by the type and effect system (Section 4). In Section 5, we describe how the types can be automatically inferred. Section 6 details some highlights of the Type-Based Java String Analyzer implementation.

2 The Programming Guideline

In the SAP NetWeaver Platform, the SAP Output Encoding Framework provides XSS prevention facilities for programs that generate HTML code and have access to certain untrusted "user input" strings, like information coming from a GET request. By encoding or "sanitizing" such user-supplied input before rendering it, any inserted scripts are prevented from being transmitted in executable form. To prevent XSS attacks, The following programming guideline for a correct framework usage is specified, in which different cases need to be distinguished.

1. When a user string is output within HTML tags, a function `escapeToHtml` should be applied for output encoding.
2. When a user string is output in a JavaScript context, a function `escapeToJs` should be applied for output encoding.

```
public void doGet(HttpRequest request, SecureAPI api) {
  String input = request.getInputParameter();
  // -- case 1: HTML embedding --
  String s = "<body>" + api.escapeToHtml(input) + "</body>";
  api.output(s);
  // -- case 2: JavaScript embedding --
  if (showAlert) {
    api.output("<script>");
    api.output(" alert('" + api.escapeToJs(input) + "');");
    api.output("</script>");
  }
}
```

Fig. 1. Example program

The functions are provided by the framework; concrete implementations could for instance remove all HTML tags or all quotation marks from the strings. Due to limited space, we leave away two other embedding cases that apply to HTML attribute values, and the fact that the methods come in overloaded versions for different use cases. For more detailed information about the guideline with respect to the usage of the SAP Output Encoding Framework, please refer to its documentation [13].

2.1 Formalization of the Programming Guideline

We now make the above guideline more precise, and illustrate this with the program in Figure 1, which shall also serve as a running example for this paper.

We assume that untrusted user strings originate in the return value of a method `getInputParameter`. All strings that are derived from these return values by string operations are also considered unsafe. The only string operation we consider is concatenation with the + operator. We assume an interface `SecureAPI` that models the framework and provides the two sanitization functions `escapeToHtml` and `escapeToJs`, as well as an output function `output`.

Before being passed to `output`, any unsafe string must be passed to one of the two sanitization functions: when the string is embedded somewhere between "<script>" and "</script>", `escapeToJs` must be used, otherwise, one shall use `escapeToHtml`. The example program satisfies the guideline, because the correct sanitization function for `input` is applied depending on where it is embedded.

Although the guideline may appear relatively simple, it already imposes a number of requirements for the analysis. It is not sufficient to approximate possible string values, as the trustworthiness of a string cannot be solely derived from its value: The same string value can be either a trusted literal or a malicious piece of injected code. On the other hand, a pure dataflow analysis is not enough, as the choice of the sanitization depends on triggers like <script> literals.

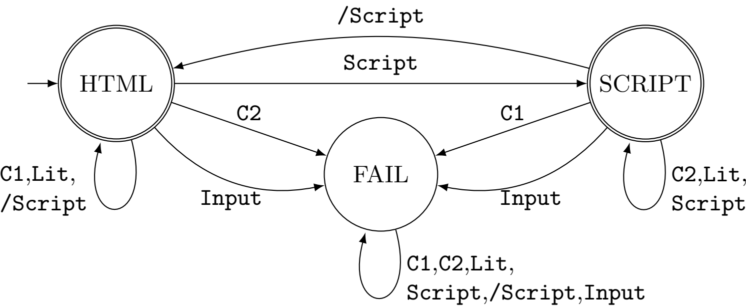

Fig. 2. Sample policy automaton

2.2 Output Traces and Policy Automaton

We classify the strings in the program according to their contents and origin:

- all strings coming from `getInputParameter` are assigned the class `Input`;
- `escapeToHtml` returns `C1`-classified strings, whereas `escapeToJs` returns `C2`-classified strings;
- literals have the classification `Lit`, except for the strings "<script>" and "</script>", which are classified `Script` and `/Script`, respectively.

Concatenating these strings and passing them to the `output` function define the *output trace* of the program, which are words over the alphabet of classifications $\Sigma = \{\texttt{Lit}, \texttt{C1}, \texttt{C2}, \texttt{Script}, \texttt{/Script}, \texttt{Input}\}$. Our program generates two traces:

1. $\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit} \cdot \texttt{Script} \cdot \texttt{Lit} \cdot \texttt{C2} \cdot \texttt{Lit} \cdot \texttt{/Script}$ if $\texttt{showAlert} = \texttt{true}$
2. $\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit}$ if $\texttt{showAlert} = \texttt{false}$

The formalized guideline now requires that all output traces are accepted by the finite state machine (*policy automaton*) given in Figure 2. The machine contains accepting states for two modes, such that "normal mode" accepts `C1`, but not `C2` strings, while "script mode" does the reverse. A switch to "script mode" occurs whenever a `Script` string is encountered, and back to "normal mode" at `/Script`. All other cases lead to a special inescapable fail state, e.g. whenever an `Input` string is encountered. A trace is accepted if it leads to an accepting state. The machine thus accepts the traces of the example program above, but not e.g. $\texttt{Script} \cdot \texttt{C1} \cdot \texttt{/Script}$, which means the wrong sanitization function is used, or `Input`, which means a string from `getInputParameter` has been directly output, and hence has not been sanitized at all.

The example machine is kept rather simple for presentation purposes. In practice, one could use extended machines, e.g. to detect mode switches for `output("<scr"+"ipt>")`, or to handle the mentioned other sanitization cases. We could also allow a nondeterministic automaton, which would then be determinized using powerset construction. However, the policy itself, i.e. the decision whether a given trace is accepted, will always be deterministic.

We now factor the infinite set of traces by behavioural equivalence with respect to the policy automaton, and in this way obtain a finite set of equivalence classes carrying a monoid structure. Formally, let $G = (Q, q_0, \delta, F)$ be the automaton with a set of states Q, an initial state q_0, accepting states F, and a transition function δ. Two words w_1 and w_2 are equivalent if they have the same effect in each state: $w_1 \cong w_2 \iff \forall q \in Q.\ \delta(q, w_1) = \delta(q, w_2)$. The equivalence class to which a word w belongs is denoted by $[w]$. Concatenation is defined on classes by $[w_1] \cdot [w_2] = [w_1 \cdot w_2]$. Note that $[\epsilon]$ is the neutral element. The subset $\mathsf{Allowed}$ denotes those equivalence classes that contain words accepted by G.

The example automaton shown above has the following associated monoid: $Mon = \{[\mathtt{Lit}], [\mathtt{C1}], [\mathtt{C2}], [\mathtt{Script}], [\mathtt{/Script}], [\mathtt{Input}], [\mathtt{C1}{\cdot}\mathtt{Script}], [\mathtt{C2}{\cdot}\mathtt{/Script}]\}$. All of these eight classes have a different effect on the automaton, and there are no more classes. The neutral element is $[\varepsilon] = [\mathtt{Lit}]$; the set of accepted classes is $\mathsf{Allowed} = \{[\mathtt{Lit}], [\mathtt{C1}], [\mathtt{Script}], [\mathtt{/Script}], [\mathtt{C1} \cdot \mathtt{Script}]\}$.

We also define a function $\mathsf{litword}$ that specifies the word $w \in \Sigma^*$ for a given string literal. For our example program, we have $\mathsf{litword}$(“<script>”)=$\mathtt{Script}$, $\mathsf{litword}$(“</script>”)=$\mathtt{/Script}$, and $\mathsf{litword}(str)$=$\mathtt{Lit}$ for all other literals str.

We assume the designer of the security guideline formalizes their requirements in the form of a finite state machine, and computes the associated monoid. Our type system is parametric with respect to a given monoid.

3 The FJEUS Language

FJEUS is a formalized and downsized object-oriented language that captures those aspects of Java that are interesting for our analysis: objects with imperative field updates, and strings. The language is an extension of FJEU [14] with strings, which itself extends Featherweight Java (FJ) [15] with side effects on a heap.

3.1 Syntax

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

variables: $x, y \in Var$ fields: $f \in Fld$ string literals: $str \in Str$
classes: $C, D \in Cls$ methods: $m \in Mtd$

$$\begin{aligned} Expr \ni e ::= {} & x \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ x_1 = x_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \\ & \mathtt{null} \mid \mathtt{new}\ C \mid x.f \mid x_1.f{:=}x_2 \mid x.m(\overline{x}) \mid "str" \mid x_1 + x_2 \end{aligned}$$

For the sake of simplicity we omit other primitive data types and type casts, and require programs to be in let normal form. The somewhat unusual equality conditional construct is included to have reasonable if-then-else expressions without relying on booleans. The language features string literals, and a concatenation + as the only string operation. An overlined term $\overline{x}$ stands for an ordered sequence.

An FJEUS program $P = (\preceq, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ defines the following: $\preceq \in \mathcal{P}(Cls \times Cls)$ is the subclass relation; $D \preceq C$ means D is a subclass of C.

The functions $\mathit{fields} \in \mathit{Cls} \to \mathcal{P}(\mathit{Fld})$, $\mathit{methods} \in \mathit{Cls} \to \mathcal{P}(\mathit{Mtd})$ specify for each class C its fields and methods. A method table $\mathit{mtable} \in \mathit{Cls} \times \mathit{Mtd} \rightharpoonup \mathit{Expr}$ gives for each method of a class its implementation, i.e. the FJEUS expression that forms the method's body. We assume the formal argument variables of a method m are named x_1^m, x_2^m, etc., besides the implicit and reserved variable this. Only these variables may occur freely in the body of m. Alternatively, the implementation may be given directly in form of a big-step semantic relation; we call such methods *external*. A number of well-formedness conditions are imposed on these functions to ensure the usual class inheritance properties; details are given in the appendix. From now on, we assume a fixed well-formed program P.

3.2 Instrumented String 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. The only kinds of values in FJEUS are object locations and null. We distinguish two kinds of objects: *ordinary objects* contain a class identifier and a valuation of the fields, while *string objects* are immutable character sequences tagged with a word w over the alphabet Σ. The state model is summarized by the following table:

locations:	$l \in \mathit{Loc}$	stores:	$s \in \mathit{Var} \rightharpoonup \mathit{Val}$
values:	$v \in \mathit{Val} = \mathit{Loc} \uplus \{\mathit{null}\}$	heaps:	$h \in \mathit{Loc} \rightharpoonup \mathit{Obj} \uplus \mathit{SObj}$
string objects:	$\mathit{SObj} = \Sigma^* \times \mathit{Str}$	objects:	$\mathit{Obj} = \mathit{Cls} \times (\mathit{Fld} \rightharpoonup \mathit{Val})$

The FJEUS semantics is defined as a big-step relation $(s, h) \vdash e \Downarrow v, h' \;\&\; w$. It means that the expression e evaluates in store s and heap h to the value v and modifies the heap to h', generating an output trace (word) $w \in \Sigma^*$.

Figure 3 shows some of the rules that define the operational semantics. We only discuss the parts that are related to strings or traces. For `let` constructs, the output traces of the subexpressions are simply concatenated. The trace of a method body execution is also the trace of the method call. String literals cause the creation of a new string object in the heap, tagged with the word given by $\mathsf{litword}$. Since a literal does not produce any output, we have the empty trace ϵ here. A concatenated string $x_1 + x_2$ is tagged by concatenating the tags of the original strings. Additional functionality like string sanitization and output is provided by external methods. Implementations for the external methods used in our example, along with the full rule system, can be found in the appendix.

We call the semantics "instrumented", because the tags attached to the string objects are imaginary and do not exist during the actual program execution. Rather, they are used here for a rigorous definition of the programming guideline. The tags have an intensional meaning which is defined either by $\mathsf{litword}$ in the case of literals, or by the semantics of external methods. We assume these methods use the "correct" tags, e.g. `getInputParameter`() returns an `Input`-tagged string.

$$\frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \;\&\; w_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \;\&\; w_2}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2 \;\&\; w_1 \cdot w_2}$$

$$\frac{s(x) = l \quad h(l) = (C, _) \quad |\overline{x^m}| = |\overline{y}| = n \qquad s' = [this \mapsto l] \cup [x_i^m \mapsto s(y_i)]_{1 \le i \le n} \qquad (s', h) \vdash mtable(C, m) \Downarrow v, h' \;\&\; w}{(s,h) \vdash x.m(\overline{y}) \Downarrow v, h' \;\&\; w}$$

$$\frac{l \notin dom(h) \qquad w = \mathsf{litword}(str) \qquad h' = h[l \mapsto (w, str)]}{(s,h) \vdash \text{"}str\text{"} \Downarrow l, h' \;\&\; \epsilon}$$

$$\frac{h(s(x_1)) = (w_1, str_1) \qquad h(s(x_2)) = (w_2, str_2) \qquad l \notin dom(h) \qquad h' = h[l \mapsto (w_1 \cdot w_2, str_1 \cdot str_2)]}{(s,h) \vdash x_1 + x_2 \Downarrow l, h' \;\&\; \epsilon}$$

Fig. 3. Operational semantics of FJEU

4 Type and Effect System

Our analysis is a type and effect system that is parametric with respect to a given policy automaton. Whenever a program is typable, it means the programmer has followed the security guideline that the automaton describes. Untypable programs either violate the guideline, or the type system is not expressive enough to show that the guideline has been followed.

4.1 Refined String Types and Class Tables

The distinction of ordinary and string objects is mirrored in the type system:

$$Typ \ni \tau, \sigma \;::=\; C \mid \texttt{String}_U \qquad \text{where} \quad U \subseteq Mon$$

A value typed with $\texttt{String}_U$ intuitively means that it is a location that refers to a string object that is tagged with a word w such that $[w] \in U$. We use subsets of Mon rather than single monoid elements to account for joining branches of conditionals (including the conditionals implicit in dynamic dispatch).

A class table (A, M) models Java's class member types. The *field typing* $A : (Cls \times Fld) \rightharpoonup Typ$ assigns to each class C and field $f \in fields(C)$ the type of the field. The type is required to be invariant with respect to subclasses of C.

The *method typing* $M : (Cls \times Mtd) \rightharpoonup \mathcal{P}(Typ^* \times Typ \times \mathcal{P}(Mon))$ assigns to each class C and each method $m \in methods(C)$ an unbounded number of *method types* $(\overline{\sigma}, \tau, U)$, from now on written $\overline{\sigma} \xrightarrow{U} \tau$, which specify the types of the formal argument variables and of the result value, as well as the possible effects of the method (explained below). All method types assigned to a method must have the same underlying unannotated Java signature, but the string type refinements as well as the method effect may differ. This enables infinite polymorphic method types, as far as the refinements to the Java type systems are concerned. For every method type in $M(C, m)$ and each subclass $C' \preceq C$, there must be an improved method type in $M(C', m)$, where improved means it is contravariant in the argument types, covariant in the result class, and has a smaller effect set.

The polymorphism makes it possible to use a different type at different invocation sites of the same method, or even at the same invocation site in different

$$\frac{\Gamma \vdash e_1 : \tau \,\&\, U \qquad \Gamma, x:\tau \vdash e_2 : \tau' \,\&\, U'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau' \,\&\, UU'} \qquad \frac{\overline{\sigma} \xrightarrow{U} \tau \in M(C,m)}{\Gamma, x:C, \overline{y}:\overline{\sigma} \vdash x.m(\overline{y}) : \tau \,\&\, U}$$

$$\frac{\mathsf{litword}(str) = w}{\Gamma \vdash \text{"}str\text{"} : \texttt{String}_{\{[w]\}} \,\&\, \{[\varepsilon]\}} \qquad \frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\varepsilon]\}}$$

Fig. 4. FJEUS Type System (extract)

type derivations. Please refer to the appendix for an example program where polymorphic method types improve the precision of the analysis of the method.

4.2 Typing Rules

The declarative typing judgement takes the form $\Gamma \vdash e : \tau \,\&\, U$ where e is an expression, Γ maps variables (at least those in e) to types, τ is a type, and U is a subset of Mon. The meaning is that if the values of the variables comply with Γ and the evaluation of e terminates successfully then the result complies with τ, and the output written during this evaluation will belong to one of the classes in U. In particular, if $U \subseteq \mathsf{Allowed}$ then e adheres to the guideline. It suffices to perform this latter check for an entry point such as the "main" method.

Figure 4 only shows some of the typing rules; the full type system can be found in the appendix. The `let` rule takes into account that first the effects of expression e_1 take place, and then the effects of expression e_2. For the concatenation of the subeffects we define $UU' = \{[w \cdot w'] \mid [w] \in U, [w'] \in U'\}$. For method calls, it suffices to choose one method type from $M(C,m)$. The type annotation for string literals relies on $\mathsf{litword}$. The type of a concatenated string is defined by concatenating the monoid elements of the two initial string types.

An FJEUS program $P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ is *well-typed* if for all classes C, methods m, and method types $\overline{\sigma} \xrightarrow{U} \tau \in M(C,m)$, one can derive the typing judgement $[\mathit{this} \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i \in \{1,\ldots,|\overline{x^m}|\}} \vdash \mathit{mtable}(C,m) : \tau \,\&\, U$.

The polymorphic method types make the 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 different derivations, different type annotations may be chosen for new string objects and for called methods. The inference algorithm later uses context-sensitive restrictions to determine the (finite) set of suitable types for each method.

4.3 External Methods

As previously mentioned, external methods are not defined syntactically by *mtable*, but by providing the semantics directly. We can nevertheless assign method types for them, which then act as *trusted signatures*, such that the

methods are considered well-typed even though no type derivation is provided. This enables the specification of trusted types for methods of the security API.

For our running example, we assume there are two classes `SecureAPI` and `HttpRequest` that contain external methods with the signatures shown below. In particular, `output` has exactly the effect given by the refinement of the string argument; exploiting polymorphism, we assign a method type for each $U \subseteq \mathit{Mon}$.

$$M(\texttt{HttpRequest}, \texttt{getInputParameter}) = \{() \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{Input}]\}}\}$$
$$M(\texttt{SecureAPI}, \texttt{escapeToHtml}) = \{\texttt{String}_{\{[\texttt{Input}]\}} \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{C1}]\}}\}$$
$$M(\texttt{SecureAPI}, \texttt{escapeToJs}) = \{\texttt{String}_{\{[\texttt{Input}]\}} \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{C2}]\}}\}$$
$$M(\texttt{SecureAPI}, \texttt{output}) = \{\texttt{String}_U \xrightarrow{U} \texttt{Void} \mid U \subseteq \mathit{Mon}\}$$

4.4 Interpretation of the Typing Judgement

We now give a formal interpretation of the typing judgement in form of a soundness theorem. It relies on a *heap typing* $\Sigma : \mathit{Loc} \rightharpoonup \mathit{Cls} \uplus \mathit{Mon}$ that assigns to each heap location l an upper bound of the actual class found at l for ordinary objects, or a monoid element that matches the tag for string objects. Heap typings are a standard practice in type systems [11] to avoid the need for a co-inductive well-typedness definitions in the presence of cyclic heap structures.

We just briefly describe how heap typings are used for the soundness statement here; the appendix contains a complete definition. We define a typing judgment $\Sigma \vdash v : \tau$, which means that according to heap typing Σ, the value v may be typed with τ. The judgment is lifted point-wise to stores and variable contexts: $\Sigma \vdash s : \Gamma$. The relation $\Sigma \vdash h$ establishes the connection to the heap: it asserts that for all locations l, the type $\Sigma(l)$ actually describes the object $h(l)$.

The interpretation of the judgement $\Gamma \vdash e : \tau \,\&\, U$ states that whenever a well-typed program is executed on a heap that is well-typed with respect to some typing Σ, then the final heap after the execution is well-typed with respect to some heap typing $\Sigma' \sqsupseteq \Sigma$ that is possibly larger to account for new objects that may have been allocated during the program execution.

Theorem 1 (Soundness). *Fix a well-typed program P. For all $\Sigma, \Gamma, \tau, s, h, e, v, h', w$ such that $\Gamma \vdash e : \tau \,\&\, U$ and $\Sigma \vdash s : \Gamma$ and $(s, h) \vdash e \Downarrow v, h' \,\&\, w$ and $\Sigma \vdash h$, there exists some $\Sigma' \sqsupseteq \Sigma$ such that $\Sigma' \vdash v : \tau$ and $\Sigma' \vdash h'$ and $[w] \in U$.*

The proof of the soundness theorem can be found in the appendix. It follows that the typability relation proves adherence to the programming guideline:

Corollary 1. *Let P be a well-typed FJEUS program. Let* `main` *be a method which takes no arguments, serves as the entry point of P, and has the implementation e. If $\vdash e : \tau \,\&\, U$ can be derived and $U \subseteq \mathsf{Allowed}$, then any output trace of the program is described by $\mathsf{Allowed}$. By definition of $\mathsf{Allowed}$, the trace is accepted by the policy automaton, thus the program follows the programming guideline.*

```
Void doGet(HttpRequest request, SecureAPI api)
  let input = request.getInputParameter() in
  let s = "<body>" + api.escapeToHtml(input) + "</body>" in
  api.output(s)
```

Fig. 5. Example program in FJEUS

5 Automatic Type Inference

Since FJEUS formalizes the core of Java, we consider programs that are completely annotated with basic class type information, as is standard in Java programs. We now present an inference algorithm that automatically computes the refinements, i.e. the annotations for the `String` types as well as the effects.

5.1 Algorithmic Type Checking

The type system from Section 4 is transformed into a syntax-directed version with typing judgements $\Gamma \; ; \; z \; \vdash \; e \; \rightrightarrows \; \tau \;\&\; U$. It suggests an algorithm that takes a type environment Γ, a context z (explained below) and an expression e, and computes the type τ and the effect U. The complete rules are given in the appendix. They are a specialization of the declarative system and are thus sound.

The algorithmic type system depends on an annotated class table. Therefore, we create a set variable $U \subseteq \mathit{Mon}$ for each declared `String` field, for each declared `String` method argument or return value, as well as for each method effect. The side conditions on these variables U in the type system are collected as *set constraints*, which can then be solved by an external set constraint solver.

Since we are interested in inferring polymorphic method types, the question arises how many different types should be computed for a method. We propose a *context-sensitive* analysis where types are distinguished according to a call context from a finite set Cxt. Methods are analysed for a given context $z \in \mathit{Cxt}$. Whenever a submethod is called, a context transfer function ϕ is used to obtain a new context z' for which a type for called method is to be derived, if not already done. As Cxt is finite, the analysis will eventually terminate. In a sense, both Cxt and ϕ are a finite abstraction of the control flow graph of the execution.

Choosing a call context is a trade-off between precision and efficiency. Following our earlier work [16], we leave the system parametric in Cxt and ϕ, so that it can be flexibly instantiated with different kinds of context sensitivity [17–19].

5.2 Typing the Example Program

We now show how the types of the example program (Figure 1) would be inferred, thereby showing that the program indeed adheres to the programming guideline. The FJEUS version of the first half of the program is shown in Figure 5.

The algorithm infers possible types and effects for all methods, and one can then check that the inferred effect of a top-level method, e.g. `doGet` or `main`, only

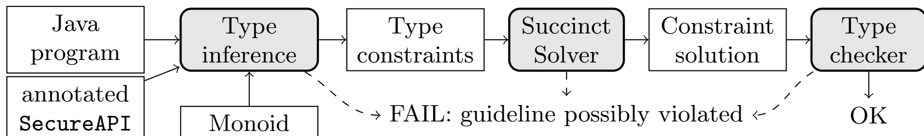

Fig. 6. Overview of the TJSA tool

contains classes from the set Allowed. The inference works as follows: the external type for `getInputParameter` gives $\texttt{String}_{\{[\texttt{Input}]\}}$ as the type for `input`. Thus, we find a matching method type for the call to `escapeToHtml`, returning a string of type $\texttt{String}_{\{[\texttt{C1}]\}}$. For the literals "<body>" and "</body>", the function litword gives the class [`Lit`]. Therefore, the concatenation produces for `s` a string type refined with $[\texttt{Lit}] \cdot [\texttt{C1}] \cdot [\texttt{Lit}] = [\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit}] = [\texttt{C1}]$. We can choose the respective type for `output` and get [`C1`] as the output effect of `doGet`. This is a subeffect of Allowed, hence the method indeed follows the guideline.

If the programmer had violated the guideline by using the wrong sanitization `escapeToJs`, the output effect would have been $[\texttt{Lit} \cdot \texttt{C2} \cdot \texttt{Lit}] = [\texttt{Input}]$, which is not in Allowed. For a detailed untypable program, as well as a program that requires context-sensitivity, please refer to the examples provided with the implementation.

6 Implementation

We have transferred string type refinements and effects to the Java language, extended the Java type system accordingly, and implemented the context-sensitive type inference in a tool called *Type-Based Java String Analyzer* (TJSA). A live demo and a documentation can be found on our website [20], where the analyzer can be tried out on several provided example programs, or on custom Java code.

The tool is based on fjavac [21], a Java compiler implemented in OCaml. We have extended the standard Java type checker with refined `String` types and method output effects. In the syntax, the programmer may specify this extended type information using certain Java annotations. The `SecureAPI` class from Section 4.3 can thus be given as an annotated Java interface.

Figure 6 gives a brief overview of the way TJSA works. Given the `SecureAPI` signature, the *unannotated* example Java program from Figure 1, and the monoid from Section 2, TJSA can fully automatically infer the missing string type refinements and output effects. It generates a variable for each string type or method effect whenever no annotation can be found in the program. The analyzer collects all set constraints for these variables according to the algorithmic type system, and solves them with the Succinct Solver tool [22]. To verify the found solution, TJSA finally checks the validity of the derivation with the inferred types. The result and the meaning of (un)typability is clearly communicated to the user.

One may add annotations by hand to support the inference or to enforce type checks, but this is generally not required. This leaves us confident that existing

code using (an annotated version of) the SAP Output Encoding Framework can be verified without modifications for compliance with the guideline.

The typing algorithm works linearly on the program structure and collects constraints on type variables parametrized by contexts, therefore the main complexity aspect lies in solving the constraints. We observe that the number of type variables is bounded by the size of the program and the context set Cxt, and the possible values for each variable is exponential in the size of the monoid Mon. Apart from that, we have not yet performed an extensive tool evaluation, as our main goal was to develop the key ideas of the analysis, focusing on correctness.

7 Conclusion and Related Work

We have shown that programming guidelines are a type of security policy that addresses security vulnerabilities at the level of coding: expert knowledge on the prevention of attacks is condensed into simple principles that are easy to implement for the programmer. We have argued that type systems are a suitable form to enforce programming guidelines, as the programmer is familiar with types, their behaviour is predictable, and the correctness is easy to maintain thanks to the separation of the declarative typability definition from the algorithmic type inference task. Also, the type inference itself is given as a rule-based judgement relying on a constraint solver whose results can then be independently verified.

In particular, we have focused on a concrete programming guideline for the correct use of sanitization functions of the SAP Output Encoding Framework to prevent cross-site scripting attacks. The guideline has been formalized by defining valid output traces, and we have given a type string system that computes and verifies such traces. As mentioned before, the type system is parametric with respect to the policy automaton. Other string-related programming guidelines that can be formalized in such a way are thus readily verifiable by our system. Indeed, it would be interesting to identify existing sanitization frameworks that are suitable for a verification with our system. In this regard, a recent formal study of common XSS sanitization mechanisms [23] complements our work.

We notice that once the guideline has been formalized with the instrumented semantics, enforcement of the guideline is also within the reach of other string analysis methods such as [24–26, 8]. Comparing the precision of our analysis with the string analysis in [24], we note that the latter approach is geared towards the approximation of string contents using context-free languages, whereas we are only interested in behaviour of strings with respect to the given policy automaton. On the other hand, our analysis incorporates interprocedural aspects via polymorphism and context sensitivity. Most importantly, our analysis is type-based with the advantageous aspects described in Section 1.3.

Finite automata have been proposed to express policies for resource usage events, and type-and-effect systems have been used to approximate events generated by a program [27, 28]. However, the validity of inferred event histories with respect to an automaton has not been verified directly with type systems be-

fore. Nevertheless, it seems promising to elaborate to what extent the mentioned approaches can be used to formalize and verify guidelines for secure coding.

The precision of the analysis could be improved by refining class types with *regions*, as outlined in the appendix, and presented in more detail in our previous work [16]. Such an object-sensitive extension enables the use of different field types to different objects of the same class.

The tool applies the concepts of the theoretical type system to Java, and informally does so even for Java language features not present in FJEUS. A more formal connection between the languages, and the extension with Java features such as exceptions and concurrency, provide further research opportunities.

Our main medium-term goal, however, is to look at programming guidelines for security in general, and to enforce them with a type-based analysis. This may involve an even tighter integration of techniques from static analysis with type systems. The underlying principles will still be correctness of the analysis, and an implementation that is easy to use and supports the programmer.

References

1. Open Web Application Security Project: The OWASP Application Security Verification Standard Project. http://www.owasp.org/index.php/ASVS
2. SAP Blog / A. Wiegenstein: A short story about Cross Site Scripting. http://www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/2422
3. Patrick Hildenbrand: Guard your web applications against XSS attacks: Output encoding functionality from SAP. SAP Insider **8**(2) (2007)
4. Open Web Application Security Project: The OWASP ten most critical web application security risks. http://owasptop10.googlecode.com/
5. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: 2006 IEEE Symp. on Security and Privacy (SP '06), Washington, DC, USA, IEEE Computer Society (2006) 258–263
6. Wikipedia: Cross-site scripting. http://en.wikipedia.org/w/index.php?title=Cross-site_scripting&oldid=417581017 (2011) [Online; accessed 14-March-2011].
7. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: 33rd Symposium on Principles of Programming Languages (POPL 2006), Charleston, SC, ACM Press New York, NY, USA (January 2006) 372–382
8. Crégut, P., Alvarado, C.: Improving the Security of Downloadable Java Applications With Static Analysis. Electr. Notes Theor. Comp. Sci. **141**(1) (2005) 129–144
9. Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In: Conf. on Prog. Lang. Design and Implementation (PLDI 2007), San Diego, CA, ACM Press New York, NY, USA (June 2007)
10. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: 14th USENIX Security Symposium (SSYM'05), Berkeley, CA, USA, USENIX Association (2005) 18–18
11. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, Berlin Heidelberg (1999)
13. SAP AG: SAP NetWeaver 7.0 Knowledge Center. http://help.sap.com/content/documentation/netweaver/

14. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: 16th European Symp. on Prog. (ESOP). Volume 3924 of LNCS., Springer (2006) 22–37
15. 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 (1999)
16. Beringer, L., Grabowski, R., Hofmann, M.: Verifying pointer and string analyses with region type systems. In: Logic for Prog., Artif. Int., and Reasoning (LPAR). Volume 6355 of Lecture Notes in Computer Science., Springer-Verlag (2010) 82–102
17. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
18. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Conf. on Programming language design and implementation (PLDI'94), New York, NY, USA, ACM (1994) 242–256
19. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. SIGPLAN Not. **39**(6) (2004) 131–144
20. Grabowski, R.: Type-Based Java String Analysis. http://jsa.tcs.ifi.lmu.de/ (2011)
21. Tse, S., Zdancewic, S.: Fjavac: a functional Java compiler. http://www.cis.upenn.edu/∼stevez/stse-work/javac/index.html (2006)
22. Nielson, F., Nielson, H.R., Seidl, H.: A succinct solver for alfp. Nordic J. of Computing **9** (December 2002) 335–372
23. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In: 16th European Symposium on Research in Computer Security (ESORICS). (2011)
24. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: 10th Int. Static Analysis Symp. (SAS '03). Volume 2694 of Lecture Notes in Computer Science., Springer-Verlag (June 2003) 1–18
25. Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An interactive tool for analyzing embedded sql queries. In: 8th Asian conference on Programming languages and systems. APLAS'10, Berlin, Heidelberg, Springer-Verlag (2010) 131–138
26. Tabuchi, N., Sumii, E., Yonezawa, A.: Regular expression types for strings in a text processing language. Electr. Notes Theor. Comput. Sci. **75** (2002)
27. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. **31** (August 2009) 23:1–23:43
28. Skalka, C., Smith, S.: History effects and verification. In: Asian Programming Languages Symposium. (November 2004)

A Operational semantics of FJEUS

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

$$\begin{array}{rl} \text{subclass relation:} & \prec \in \mathcal{P}(\mathit{Cls} \times \mathit{Cls}) \\ \text{field list:} & \mathit{fields} \in \mathit{Cls} \to \mathcal{P}(\mathit{Fld}) \\ \text{method list:} & \mathit{methods} \in \mathit{Cls} \to \mathcal{P}(\mathit{Mtd}) \\ \text{method table:} & \mathit{mtable} \in \mathit{Cls} \times \mathit{Mtd} \rightharpoonup \mathit{Expr} \\ \text{FJEU program:} & P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable}) \end{array}$$

FJEUS 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 FJEUS expression that forms the method's body. We assume that the formal argument variables of a method m are named x_1^m, x_2^m, etc., besides the implicit and reserved variable *this*. Only these variables may occur freely in the body of m. A method table is well-formed if for all $m \in \mathit{methods}(C)$, either $\mathit{mtable}(C, m)$ is defined, or the implementation is given directly in form of a big-step semantic relation in the case of external methods. Implementation may be overridden in subclasses for the same number of formal parameters. For simplicity, we do not include overloading. In the following, we assume a fixed well-formed FJEUS program P.

The complete rules for the operational semantics can be found in Figure 7. A premise involving a partial function, like $s(x) = l$, always implies the side condition $x \in \mathit{dom}(s)$. The rules for variables, `null`, and the conditional are standard. For `let`, we concatenate the output traces (words) of the two expressions. A new object is allocated at a fresh location with all fields set to *null*. A field read access returns the field contents, while a field write access updates the heap accordingly (and also evaluates to the written value). At a method call, a new store is created, consisting of a special variable *this* and of the method parameters initialized with the values of the passed arguments. The return value, final heap and output trace of the method execution are also the result of the call. In the rule for string literals, we rely on litword to tag the new string object. The tagging of strings is a homomorphism with respect to string concatenation. Since the non-recursive rules do not produce any output, we have the empty trace ϵ there.

$$\frac{}{(s,h) \vdash x \Downarrow s(x), h \ \&\ \epsilon} \qquad \frac{}{(s,h) \vdash \texttt{null} \Downarrow \mathit{null}, h \ \&\ \epsilon}$$

$$\frac{s(x) = s(y) \qquad (s,h) \vdash e_1 \Downarrow v, h' \ \&\ w}{(s,h) \vdash \texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v, h' \ \&\ w}$$

$$\frac{s(x) \neq s(y) \qquad (s,h) \vdash e_2 \Downarrow v, h' \ \&\ w}{(s,h) \vdash \texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v, h' \ \&\ w}$$

$$\frac{\begin{array}{c}(s,h) \vdash e_1 \Downarrow v_1, h_1 \ \&\ w_1 \\ (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \ \&\ w_2\end{array}}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2 \ \&\ w_1 \cdot w_2} \qquad \frac{\begin{array}{c}l \notin \mathit{dom}(h) \\ F = [f \mapsto \mathit{null}]_{f \in \mathit{fields}(C)}\end{array}}{(s,h) \vdash \texttt{new}\ C \Downarrow l, h[l \mapsto (C, F)] \ \&\ \epsilon}$$

$$\frac{s(x) = l \qquad h(l) = (_, F)}{(s,h) \vdash x.f \Downarrow F(f), h \ \&\ \epsilon} \qquad \frac{\begin{array}{c}s(x) = l \qquad h(l) = (C, F) \\ h' = h[l \mapsto (C, F[f \mapsto s(y)])]\end{array}}{(s,h) \vdash x.f{:=}y \Downarrow s(y), h' \ \&\ \epsilon}$$

$$\frac{\begin{array}{c}s(x) = l \qquad h(l) = (C, _) \qquad |\overline{x^m}| = |\overline{y}| = n \\ s' = [\mathit{this} \mapsto l] \cup [x_i^m \mapsto s(y_i)]_{i \in \{1,\ldots,n\}} \\ (s', h) \vdash \mathit{mtable}(C, m) \Downarrow v, h' \ \&\ w\end{array}}{(s,h) \vdash x.m(\overline{y}) \Downarrow v, h' \ \&\ w} \qquad \frac{\begin{array}{c}l \notin \mathit{dom}(h) \\ h' = h[l \mapsto (\mathsf{litword}(\mathit{str}), \mathit{str})]\end{array}}{(s,h) \vdash \text{"}\mathit{str}\text{"} \Downarrow l, h' \ \&\ \epsilon}$$

$$\frac{\begin{array}{c}h(s(x_1)) = ([w_1], \mathit{str}_1) \qquad h(s(x_2)) = ([w_2], \mathit{str}_2) \\ l \notin \mathit{dom}(h) \qquad h' = h[l \mapsto ([w_1 \cdot w_2], \mathit{str}_1 \cdot \mathit{str}_2)]\end{array}}{(s,h) \vdash x_1 + x_2 \Downarrow l, h' \ \&\ \epsilon}$$

Fig. 7. Complete rules of the FJEUS operational semantics

B Semantics of external methods

This section gives an example implementation of the external methods used in our application scenario to illustrate how they work.

The method `getInputParameter` models user input by returning a arbitrary string in a non-deterministic fashion. The string, however, is tagged with `Input`. The `output` method takes a string, returns *null*, and produces as the output trace the word w that the string parameter has been tagged with. The method `escapeToHtml` takes a pointer to a string object and creates a new string whose tag is `C1` and whose value has the angle brackets replaced with the corresponding HTML entities. The method `escapeToJs` works in a similar way, but escapes quotation marks and uses the `C2` tag.

$$\frac{l \notin dom(h) \qquad str \text{ arbitrary} \qquad h' = h[l \mapsto (\texttt{Input}, str)]}{(s, h) \vdash x.\texttt{getInputParameter}() \Downarrow l, h' \ \& \ \epsilon}$$

$$\frac{h(s(y)) = l \qquad h(l) = (w, str)}{(s, h) \vdash x.\texttt{output}(y) \Downarrow null, h \ \& \ w}$$

$$\frac{\begin{array}{c} h(s(y)) = (\texttt{Input}, str) \\ str' = str \text{ where } < \text{ and } > \text{ have been replaced with \< and \>} \\ l \notin dom(h) \qquad h' = h[l \mapsto (\texttt{C1}, str')] \end{array}}{(s, h) \vdash x.\texttt{escapeToHtml}(y) \Downarrow l, h' \ \& \ \epsilon}$$

$$\frac{\begin{array}{c} h(s(y)) = (\texttt{Input}, str) \\ str' = str \text{ where '' has been replaced with } \backslash\text{''} \\ l \notin dom(h) \qquad h' = h[l \mapsto (\texttt{C2}, str')] \end{array}}{(s, h) \vdash x.\texttt{escapeToJs}(y) \Downarrow l, h' \ \& \ \epsilon}$$

Fig. 8. Example semantics of external methods

C Type system

As string annotations are an over-approximation of the tags, we can easily define a subtyping relation <: based on set inclusion and the subclass relation.

$$\frac{C \preceq D}{C <: D} \qquad \frac{U \subseteq U'}{\mathtt{String}_U <: \mathtt{String}_{U'}}$$

The complete rules for the FJEUS type system can be found in Figure 9. The rule T-Sub is used to obtain weaker types for the expression. The rule T-Var looks up the type of a variable in the context Γ. In T-If, we require that both subbranches have the same type and effect. This can be obtained in conjunction with the subsumption (weakening) rule. In T-Let, we take into account that first the effects of expression e_1 take place, and then the effects of expression e_2. The overall effect is thus the concatenation of the subeffects. In T-Null, the `null` expression may have any type (class or refined string type). In the rule T-Lit, we use litword to determine the annotation of the `String` type. The rules T-New, T-Invoke, T-GetF, and T-SetF are standard. The string type annotation of a string concatenation is determined by point-wise concatenating the monoid elements of the annotations of the two initial strings (T-Concat).

$$\text{T-Sub}\ \frac{\Gamma \vdash e : \tau \,\&\, U \qquad \tau <: \tau' \qquad U \subseteq U'}{\Gamma \vdash e : \tau' \,\&\, U'} \qquad \text{T-Let}\ \frac{\Gamma \vdash e_1 : \tau \,\&\, U \qquad \Gamma, x:\tau \vdash e_2 : \tau' \,\&\, U'}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau' \,\&\, UU'}$$

$$\text{T-Var}\ \frac{}{\Gamma, x:\tau \vdash x : \tau \,\&\, \{[\varepsilon]\}} \qquad \text{T-Lit}\ \frac{\mathsf{litword}(str) = w}{\Gamma \vdash \text{"}str\text{"} : \mathtt{String}_{\{[w]\}} \,\&\, \{[\varepsilon]\}}$$

$$\text{T-New}\ \frac{}{\Gamma \vdash \mathtt{new}\ C : C \,\&\, \{[\varepsilon]\}} \qquad \text{T-Invoke}\ \frac{\overline{\sigma} \xrightarrow{U} \tau \in M(C,m)}{\Gamma, x:C, \overline{y}:\overline{\sigma} \vdash x.m(\overline{y}) : \tau \,\&\, U}$$

$$\text{T-GetF}\ \frac{A(C,f) = \tau}{\Gamma, x:C \vdash x.f : \tau \,\&\, \{[\varepsilon]\}} \qquad \text{T-SetF}\ \frac{\tau <: A(C,f)}{\Gamma, x:C, y:\tau \vdash x.f{:=}y : \tau \,\&\, \{[\varepsilon]\}}$$

$$\text{T-If}\ \frac{\Gamma \vdash e_1 : \tau \,\&\, U \qquad \Gamma \vdash e_2 : \tau \,\&\, U}{\Gamma \vdash \mathtt{if}\ x_1 = x_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, U} \qquad \text{T-Null}\ \frac{}{\Gamma \vdash \mathtt{null} : \tau \,\&\, \{[\varepsilon]\}}$$

$$\text{T-Concat}\ \frac{\Gamma(x_1) = \mathtt{String}_U \qquad \Gamma(x_2) = \mathtt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \mathtt{String}_{UU'} \,\&\, \{[\varepsilon]\}}$$

Fig. 9. The complete FJEUS type system

D Complete soundness proof

For the soundness proof, we use a heap typing $\Sigma : Loc \to (Cls \times Mon)$, and define a typing judgment for values $\Sigma \vdash v : \tau$, which means that according to heap typing Σ, the value v may be typed with τ. In particular, the information in $\Sigma(l)$ specifies the type of l.

$$\frac{}{\Sigma \vdash \mathtt{null} : \tau} \qquad \frac{\Sigma(l) = C}{\Sigma \vdash l : C} \qquad \frac{\Sigma(l) = [w]}{\Sigma \vdash l : \mathtt{String}_{[w]}} \qquad \frac{\Sigma \vdash v : \sigma \qquad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

The typing judgment of locations is lifted to stores and variable contexts as follows:

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

A heap h is *well-typed* with respect to a heap typing Σ and implicitly a field typing A, written $\Sigma \vdash h$, if the object at each location is "valid" with respect to the type predicted by Σ for that location:

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

where

$$\begin{aligned} \Sigma \vdash (C, F) : C' &\iff C \preceq C'\ \wedge\ dom(F) = fields(C)\ \wedge \\ &\qquad \forall f \in fields(C).\ \Sigma \vdash F(f) : A(C, f) \\ \Sigma \vdash (w, str) : [w'] &\iff [w] = [w'] \end{aligned}$$

A heap typing Σ' *extends* a heap typing Σ, written $\Sigma' \sqsupseteq \Sigma$, if $dom(\Sigma) \subseteq dom(\Sigma')$ and $\forall l \in dom(\Sigma).\ \Sigma(l) = \Sigma'(l)$.

With these preliminaries, we can give the proof of the soundness theorem.

Theorem 2 (Soundness Theorem). *Fix a well-typed program P. For all $\Sigma, \Gamma, \tau, s, h, e, v, h', t$ with*

$$\Gamma \vdash e : \tau \,\&\, U \quad \textit{and} \quad \Sigma \vdash s : \Gamma \quad \textit{and} \quad (s, h) \vdash e \Downarrow v, h' \,\&\, t \quad \textit{and} \quad \Sigma \vdash h$$

there exists some $\Sigma' \sqsupseteq \Sigma$ such that

$$\Sigma' \vdash v : \tau \quad \textit{and} \quad \Sigma' \vdash h' \quad \textit{and} \quad \mathsf{trclass}(t) \in U.$$

Proof. By induction over derivation of the operational semantics and the typing judgement.

We first consider the case where $\Gamma \vdash e : \tau \,\&\, U$ has been derived by the subtyping rule. By rule inversion, we get $\Gamma \vdash e : \tau' \,\&\, U'$ and $\tau' <: \tau$ and $U' \subseteq U$. With this (smaller) derivation of the typing judgement, we can apply the theorem inductively, and get $\Sigma' \vdash v : \tau'$ and $\Sigma' \vdash h'$ and $\mathsf{trclass}(t) \in U'$

for some $\Sigma \sqsupseteq {}'\Sigma$. As $\tau' <: \tau$ and $U' \subseteq U'$, we can deduce $\Sigma \vdash v : \tau$ and $\mathsf{trclass}(t) \in U$.

In the following, we assume that the typing judgement has not been derived by a subtyping rule, and continue with a case distinction over the possible forms of the big-step semantics relation.

- $(s, h) \vdash x \Downarrow s(x), h \;\&\; \epsilon$.
  Then we have $\Gamma, x : \tau \;\vdash\; x \;:\; \tau \;\&\; \{[\varepsilon]\}$, and $h' = h$ and $v = s(x)$. With $\Sigma' = \Sigma$, we get $\Sigma' \vdash h'$. As $\Sigma' \vdash s : (\Gamma, x : \tau)$, we can deduce $\Sigma' \vdash v : \tau$. Finally, $\mathsf{trclass}(\epsilon) = [\varepsilon]$.
- $(s, h) \vdash \texttt{null} \Downarrow \mathit{null}, h \;\&\; \epsilon$.
  Then we have $\Gamma \;\vdash\; \texttt{null} \;:\; \tau \;\&\; \{[\varepsilon]\}$, $v = \mathit{null}$, $h' = h$. With $\Sigma' = \Sigma$, we get $\Sigma' \vdash v : \tau$ by definition of well-typed values, and $\Sigma' \vdash h'$ follows directly from the assumption. Finally, $\mathsf{trclass}(\epsilon) = [\varepsilon]$.
- $(s, h) \vdash \texttt{new}\, C \Downarrow l, h[l \mapsto (C, F)] \;\&\; \epsilon$.
  Then $\Gamma \;\vdash\; \texttt{new}\, C \;:\; C_{\{r\}} \;\&\; \{[\varepsilon]\}$ and $v = l$ and $h' = h[l \mapsto (C, F)]$ where $l \notin \mathit{dom}(h)$ and $F = [f \mapsto \mathit{null}]_{f \in \mathit{fields}(C)}$. Since $\mathit{dom}(\Sigma) \subseteq \mathit{dom}(h)$, we have $l \notin \mathit{dom}(\Sigma)$. We choose $\Sigma' = \Sigma[l \mapsto C]$, and thus have $\Sigma' \vdash l : C$. To show $\Sigma' \vdash h'$, it suffices to show $\Sigma' \vdash h'(l) : C$. This holds trivially, as $h'(l) = (C, F)$ and $C \preceq C$ and $F(f) = \mathit{null}$ for all $f \in \mathit{dom}(F)$. Also, $\mathsf{trclass}(\epsilon) = [\varepsilon]$.
- $(s, h) \vdash \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \Downarrow v_2, h_2 \;\&\; t_1 :: t_2$.
  Then the typing judgement is $\Gamma \;\vdash\; \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \;:\; \tau \;\&\; UU'$. Then by rule inversion of the typing and semantics rule, we get $\Gamma \;\vdash\; e_1 \;:\; \tau \;\&\; U$ and $\Gamma, x : \tau \;\vdash\; e_2 \;:\; \tau' \;\&\; U'$, as well as $(s, h) \vdash e_1 \Downarrow v_1, h_1 \;\&\; t_1$ and $(s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \;\&\; t_2$. By induction, we get $\Sigma_1 \vdash v_1 : \tau$ and $\Sigma_1 \vdash h_1$ and $\mathsf{trclass}(t_1) \in U$ for some $\Sigma_1 \sqsupseteq \Sigma$. As s can include at most locations in $\mathit{dom}(\Sigma)$ and since Σ_1 is a disjoint extension of Σ, we also get $\Sigma_1 \vdash s : \Gamma$, and thereby $\Sigma_1 \vdash s[x \mapsto v_1] : \Gamma[x \mapsto \tau]$. Again by induction, we get $\Sigma' \vdash v_2 : \tau'$ and $\Sigma' \vdash h_2$ and $\mathsf{trclass}(t_2) \in U'$ for some $\Sigma' \sqsupseteq \Sigma_1$. We can transitively deduce $\Sigma' \sqsupseteq \Sigma$. Finally, we have $\mathsf{trclass}(t_1) \cdot \mathsf{trclass}(t_2) \in UU'$ by definition of monoid set concatenation, and $\mathsf{trclass}(t_1) \cdot \mathsf{trclass}(t_2) = \mathsf{trclass}(t_1 :: t_2)$ by definition of $\mathsf{trclass}$. Thus, $\mathsf{trclass}(t) \in UU'$.
- $(s, h) \vdash \texttt{if}\; x = y \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \Downarrow v, h' \;\&\; t$.
  The typing judgement is $\Gamma \;\vdash\; \texttt{if}\; x = y \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \;:\; \tau \;\&\; U$. As there are two rules for the semantics of if-then-else constructs, we either get $(s, h) \vdash e_1 \Downarrow v, h' \;\&\; t$ or $(s, h) \vdash e_2 \Downarrow v, h' \;\&\; t$ by rule inversion. Without loss of generality, we only treat the first case here. By inversion of the typing rule, we get $\Gamma \;\vdash\; e_1 \;:\; \tau \;\&\; U$. By induction, there exists a $\Sigma' \sqsupseteq \Sigma$ such that $\Sigma' \vdash v : \tau$ and $\Sigma' \vdash h'$ and $\mathsf{trclass}(t) \in U$.
- $(s, h) \vdash x.f \Downarrow F(f), h \;\&\; \epsilon$.
  Then $e = x.f$ and $h' = h$ and there is some $l \in \mathit{dom}(h)$ and some value v such that $s(x) = l$ and $h(l) = (D, F)$ and $F(f) = v$ for some class D. The typing judgement is $\Gamma, x : C \;\vdash\; x.f \;:\; \tau \;\&\; \{[\varepsilon]\}$. Rule inversion gives us $A(C, f) = \tau$. With $\Sigma' = \Sigma$, we get $\Sigma' \vdash s : (\Gamma, x : C)$, which implies $\Sigma' \vdash s(x) : C$. We also have $\Sigma' \vdash h'$, and thus by definition $\Sigma' \vdash F(f) : A(C, f)$, thus $\Sigma' \vdash v : \tau$. Also, $\mathsf{trclass}(\epsilon) = [\varepsilon]$.

– $(s,h) \vdash x.f{:=}y \Downarrow s(y), h' \;\&\; \epsilon$.

Then $e = x.f{:=}y$ and $v = s(y)$. The typing judgement is $\Gamma, x : C, y : \tau \;\vdash\; x.f{:=}y \;:\; \tau \;\&\; \{[\varepsilon]\}$. We choose $\Sigma' = \Sigma$. With $\Sigma \vdash s : (\Gamma, x : C, y : \tau)$, we have $\Sigma' \vdash v : \tau$.

With the rule of the operational semantics, there is some $l \in dom(h)$ such that $s(x) = l$ and $h(l) = (D, F)$ and $h' = h[l \mapsto (D, F[f \mapsto s(y)])]$. By $\Sigma \vdash h$, we know $D \preceq C$. We still need to show $\Sigma' \vdash h'$. Because h' and h are identical with the exception of $h'(l)(f)$, we need to show that the semantic heap relation is still preserved for the field value, i.e. that $\Sigma' \vdash s(y) : A(C, f)$. With well-formedness of the class table and the premise of the rule, we have $\tau <: A(C, f)$, hence $\Sigma' \vdash s(y) : \tau$, so $\Sigma' \vdash h'(l)(f) : \tau$. Also, $\mathsf{trclass}(\epsilon) = [\varepsilon]$.

– $(s,h) \vdash x.m(\overline{y}) \Downarrow v, h' \;\&\; t$.

By inversion of the rule for the operational semantics, we know there is a location $l \in dom(h)$ such that $s(x) = l$ and $h(l) = (D, _)$, and $(s', h) \vdash mtable(D, m) \Downarrow v, h' \;\&\; t$ where $s' = [this \mapsto l] \cup [x_i^m \mapsto s(y_i)]_{i \in \{1,\ldots,|\overline{x^m}|\}}$.

The typing judgement is $\Gamma, x : C, \overline{y} : \overline{\sigma} \;\vdash\; x.m(\overline{y}) \;:\; \tau \;\&\; U$. As $\Sigma \vdash s : (\Gamma, x : C, \overline{y} : \overline{\sigma})$, we have $\Sigma \vdash l : C$.

With $\Sigma \vdash h$, we can also infer $D \preceq C$. By inversion of the typing rule, we know typing $\overline{\sigma} \xrightarrow{U} \tau \in M(C, m)$. As $D \preceq C$ and the class table is well-formed, there is a method typing $\overline{\sigma'} \xrightarrow{U'} \tau' \in M(D, m)$ such that $\overline{\sigma} <: \overline{\sigma'}$ and $\tau' <: \tau$ and $U' \subseteq U$. From $\Sigma \vdash s : (\Gamma, x : C, \overline{y} : \overline{\sigma})$, it follows $\Sigma \vdash s(y_i) : \sigma'_i$ for $i \in \{1, \ldots, |\overline{x}^m|\}$.

As the FJEU program is well-typed, we get $\Gamma' \;\vdash\; mtable(D, m) \;:\; \tau' \;\&\; U'$ where $\Gamma' = [this \mapsto C] \cup [x_i^m \mapsto \sigma'_i]_{i \in \{1,\ldots,|\overline{x^m}|\}}$. With the facts from above, we get $\Sigma \vdash s' : \Gamma'$, so we can finally apply the theorem inductively on the derivation of the semantics and get $\Sigma' \vdash v : \tau'$ and $\Sigma' \vdash h'$ and $\mathsf{trclass}(t) \in U'$ for some $\Sigma' \sqsupseteq \Sigma$. From $\tau' <: \tau$ and $U' \subseteq U$ follows $\Sigma' \vdash v : \tau$ and $\mathsf{trclass}(t) \in U$.

– $(s,h) \vdash "str" \Downarrow l, h' \;\&\; \epsilon$.

By inversion of the rule of the operational semantics, we get $l \notin dom(h)$ and $w = \mathsf{litword}(str)$ and $h' = h[l \mapsto (w, str)]$. Also, we have $\Gamma \;\vdash\; "str" \;:\; \mathtt{String}_{\{[w]\}} \;\&\; \{[\varepsilon]\}$. We choose $\Sigma' = \Sigma[l \mapsto [w]]$, and thus have $\Sigma' \vdash l : \mathtt{String}_{\{[w]\}}$. Since $\Sigma' \vdash (w, str) : [w]$, we also have $\Sigma' \vdash h'$. Finally, $\mathsf{trclass}\epsilon = [\varepsilon]$.

– $(s,h) \vdash x_1 + x_2 \Downarrow l, h' \;\&\; \epsilon$.

By inversion of the semantics rule, we get $l \notin dom(h)$ and $h(s(x_1)) = (w_1, str_1)$ and $h(s(x_2)) = (w_2, str_2)$ and $h' = h[l \mapsto (w_1 \cdot w_2, str_1 \cdot str_2)]$. The typing rule gives $\Gamma \;\vdash\; x_1 + x_2 \;:\; \mathtt{String}_{UU'} \;\&\; [\varepsilon]$ and $\Gamma(x_1) = \mathtt{String}_U$ and $\Gamma(x_2) = \mathtt{String}_{U'}$. With $\Sigma \vdash s : \Gamma$, we have $\Sigma \vdash s(x_i) : \Gamma(x_i)$ for $i \in \{1, 2\}$. Also, from $\Sigma \vdash h$, we get $\Sigma(s(x_1)) = [w_1]$ and $\Sigma(s(x_2)) = [w_2]$. By definition of well-typed values, this means $[w_1] \in U$ and $[w_2] \in U'$. Thus, $[w_1 \cdot w_2] \in UU'$. Let $\Sigma' = \Sigma[l \mapsto [w_1 \cdot w_2]]$. It follows $\Sigma' \vdash h'$. We have $\Sigma' \vdash l : \mathtt{String}_{\{[w_1 \cdot w_2]\}}$, and, since $[w_1 \cdot w_2] \in UU'$, $\Sigma' \vdash l : \mathtt{String}_{UU'}$. Finally, $\mathsf{trclass}\epsilon = [\varepsilon]$.

E Polymorphic method types and context-sensitive analysis

Consider the program in Figure 10 (we use named type variables for illustration). Since appendLn is first called with a [Input] and then with a [C1] string, the best string type that can be inferred for ?a is {[Input], [C1]}, hence the best type for ?b is also {[Input], [C1]}, which is then the solution for the types of both x and y. The overall output effect is thus {[Input], [C1]}, hence the method is not well-typed, because a smaller effect was declared.

However, only y is actually output. It is obtained by HTML-sanitizing an input string and appending an [Lit] literal. A more precise type for y would thus be {[C1]}, which would also render the output effect well-typed. The problem is that we lose precision by assigning just one method type to appendLn, although the method is called several times with differently typed String values.

The solution is to use polymorphic method types, i.e. by assigning several method types (signatures) to appendLn, such that the best one can be chosen for each method call. In the example, we could use two different types:

$$M(C, \mathtt{appendLn}) = \{\ \mathtt{String}_{\{[\mathtt{Input}]\}} \xrightarrow{\{[\mathtt{Lit}]\}} \mathtt{String}_{\{[\mathtt{Input}]\}},\ \mathtt{String}_{\{[\mathtt{C1}]\}} \xrightarrow{\{[\mathtt{Lit}]\}} \mathtt{String}_{\{[\mathtt{C1}]\}} \}$$

To infer these types, we may identify call contexts with program lines of the method invocation. In the example program above, that means we distinguish both invocation sites of appendLn, and thus generate variables for two different signatures.

Another possible candidate for call contexts are stacks of invocation sites. In the example above, this enables the analysis to further distinguish calls to appendLn depending on the site where the outer method, test, has been called. We require, however, that the call context is finite, since the stacks may grow arbitrarily in the presence of recursive calls.

```
String_?b appendLn (s : String_?a) & {[C1]} =
    s + "\n"

Void  test() & {[C1]} =
  let input = getInputParameter() in
  let sanitized = escapeToHTML(input) in
  ...
  let x = appendLn(input) in
  let y = appendLn(sanitized) in
  output(y)
```

Fig. 10. Example program motivating polymorphic method types

F Algorithmic type system

We now present a syntax-directed form of the previous type system, from which one can directly read off an algorithm $Alg(\Gamma, e) = \tau$ that computes "from left to right" the type τ of an expression e based on a store typing Γ, similar to the approach taken by Pierce [11]. For this, we eliminate the subsumption rule, and instead specify the most precise resulting type τ for each expression. The full algorithmic type system is shown in Figure 11.

In the new system, the notion of program points is made explicit by annotating expressions with *expression labels* $i \in PP$: we write $[e]^i$ for FJEUS expressions, where e is defined as before. An FJEUS 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.

Also, we assume a given *parametrized* method typing $\hat{M} : (Cls \times Mtd) \rightharpoonup Cxt \rightharpoonup \overline{Typ} \times Typ$ replaces the previous polymorphic method typing M in the class table. It is well-formed if for all classes C and subclasses $C' \preceq C$, methods $m \in methods(C)$, and contexts $z \in Cxt$, it holds $\hat{M}(C', m)(z) <: \hat{M}(C, m)(z)$.

For the TA-Null rule, the type system needs some guidance for the choice of the type. We solve this problem by assuming that the syntax is annotated with a class: the expression $\texttt{null}_C$ means that the type system shall use the type C for that particular *null* value; the expression $\texttt{null}_{\texttt{String}}$ means the type system shall use the type $\texttt{String}_\emptyset$. The reason why we have not introduced this annotated syntax from the beginning is that the inference of class information is orthogonal to pointer analysis and the verification of region sets.

For the rule TA-If, we compute the least upper bound of two or more types. We define the join $\texttt{String}_U \vee \texttt{String}_{U'} = \texttt{String}_{U \cup U'}$ and $C_1 \vee C_2 = C$ where C is the most specific common superclass of C_1 and C_2.

We now outline a correctness proof of the algorithmic type system: For a given parametrized method typing $\hat{M}$, we define a the corresponding polymorphic method typing $M(C, m) := \bigcup_{z \in Cxt} \{\hat{M}(C, m)(z)\}$. We need to show that if $\Gamma \; ; \; z \; \vdash \; e \; \rightrightarrows \; \tau \; \& \; U$ can be derived from some $\hat{M}$ in the algorithmic system, then $\Gamma \; \vdash \; e \; : \; \tau \; \& \; U$ can be derived in the original system with respect to the corresponding method typing M. It suffices to show that the rules of the algorithmic system are derivable in the previous system:

The rule TA-If is derivable by a combination of the rules T-If and T-Sub. The rule TA-Null is a specialization of T-Null. For method calls, the type system chooses a specific method type determined by the ϕ function, which is thus a specialization of T-Invoke. All other rules as well as the definition of annotated class tables remain unchanged. We can therefore apply the soundness theorem to the algorithmic type system.

$$\text{TA-LET}\ \frac{\Gamma\ ;\ z \vdash e_1 \ \rightrightarrows\ \tau \,\&\, U \qquad \Gamma, x:\tau\ ;\ z \vdash e_2 \ \rightrightarrows\ \tau' \,\&\, U'}{\Gamma\ ;\ z \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ \rightrightarrows\ \tau' \,\&\, UU'}$$

$$\text{TA-VAR}\ \frac{}{\Gamma, x:\tau\ ;\ z \vdash x \ \rightrightarrows\ \tau \,\&\, \{[\varepsilon]\}} \qquad \text{TA-NULL}\ \frac{}{\Gamma\ ;\ z \vdash \texttt{null}_C \ \rightrightarrows\ C \,\&\, \{[\varepsilon]\}}$$

$$\text{TA-NEW}\ \frac{}{\Gamma\ ;\ z \vdash \texttt{new}\ C \ \rightrightarrows\ C \,\&\, \{[\varepsilon]\}}$$

$$\text{TA-INVOKE}\ \frac{z' = \phi(z, C, m, i) \qquad \hat{M}(C,m)(z') = \overline{\sigma}' \xrightarrow{U} \tau \qquad \overline{\sigma} <: \overline{\sigma}'}{\Gamma, x:C, \overline{y}:\overline{\sigma}\ ;\ z \vdash x.m(\overline{y}) \ \rightrightarrows\ \tau \,\&\, U}$$

$$\text{TA-GETF}\ \frac{A(C,f) = \tau}{\Gamma, x:C\ ;\ z \vdash x.f \ \rightrightarrows\ \tau \,\&\, \{[\varepsilon]\}}$$

$$\text{TA-SETF}\ \frac{\tau <: A(C,f)}{\Gamma, x:C, y:\tau\ ;\ z \vdash x.f{:=}y \ \rightrightarrows\ \tau \,\&\, \{[\varepsilon]\}}$$

$$\text{TA-IF}\ \frac{\Gamma\ ;\ z \vdash e_1 \ \rightrightarrows\ \tau_1 \,\&\, U_1 \qquad \Gamma\ ;\ z \vdash e_2 \ \rightrightarrows\ \tau_2 \,\&\, U_2}{\Gamma\ ;\ z \vdash \texttt{if}\ x_1 = x_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \ \rightrightarrows\ \tau_1 \vee \tau_2 \,\&\, U_1 \cup U_2}$$

$$\text{T-LIT}\ \frac{\mathsf{litword}(str) = w}{\Gamma\ ;\ z \vdash \text{"}str\text{"} \ \rightrightarrows\ \texttt{String}_{\{[w]\}} \,\&\, \{[\varepsilon]\}}$$

$$\text{T-CONCAT}\ \frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma\ ;\ z \vdash x_1 + x_2 \ \rightrightarrows\ \texttt{String}_{UU'} \,\&\, \{[\varepsilon]\}}$$

Fig. 11. The algorithmic FJEUS type system

G Improving the Precision with Regions

So far, ordinary objects (i.e., not `String` objects) that occur at runtime are distinguished in the type system by their static class information. We now improve the precision by further refining types for ordinary objects. For this, we assume an infinite set *Reg* of *regions* r, which are abstract memory locations. Each region stands for a zero or more concrete locations. Different regions represent disjoint sets of concrete locations; pointers to different regions thus never alias.

Class types are equipped with subsets of *Reg*. For example, a location l is typed with the *refined type* $C_{\{r,s\}}$ if it points to an object of class C or below, and if l is abstracted to either region r or region s, but no other region. (We use sets instead of single regions for the same reason as we use sets of monoid elements for string types.) The class table is refined: the field typing A assigns for each class C and each region r a type to each field f.

For example, assume we have a string buffer class with a single `String` field:

```
class StringBuf { String str; }
```

Using the refined class table, we can now distinguish the refined string type of the field `str` by the region where the StringBuf object resides:

$$A(\texttt{StringBuf}, r, \texttt{str}) = \texttt{String}_U \qquad A(\texttt{StringBuf}, s, \texttt{str}) = \texttt{String}_V$$

For the inference, we generate variables for the different region sets. The algorithmic type system gives constraints on these region set variables, which can then be solved with a constraint solver. The precision of the inference depends on the chosen set of regions, and of the choice of regions for new objects. We leave both abstract as parameters of the type system, just as we did with contexts.

Such an object-sensitive analysis complements the context-sensitive approach, and can also be made to work together. For a more detailed account on these concepts, please refer to our previous work [16].