Abstract Interpretation Plugins for Type Systems

Tobias Gedell and Daniel Hedin

Chalmers University of Technology

Abstract. The precision of many type based analyses can be significantly increased given additional information about the programs' execution. For this reason it is not uncommon for such analyses to integrate supporting analyses computing, for instance, nil-pointer or alias information. Such integration is problematic for a number of reasons: 1) it obscures the original intention of the type system especially if multiple additional analyses are added, 2) it makes use of already available analyses difficult, since they have to be rephrased as type systems, and 3) it is non-modular: changing the supporting analyses implies changing the entire type system.
Using ideas from abstract interpretation we present a method for parameterizing type systems over the results of abstract analyses in such a way that one modular correctness proof can be obtained. This is achieved by defining a general format for information transferal and use of the information provided by the abstract analyses. The key gain from this method is a clear separation between the correctness of the analyses and the type system, both in the implementation and correctness proof, which leads to a comparatively easy way of changing the parameterized analysis, and making use of precise, and hence complicated analyses. In addition, we exemplify the use of the framework by presenting a parameterized type system that uses additional information to improve the precision of exception types in a small imperative language with arrays.

1 Introduction

In the book Types and Programming Languages [14] Pierce defines a type system in the following way: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".

Pierce limits his definition to the absence of *certain* program behaviors, since many interesting (bad) behaviors cannot be ruled out statically. Well-known examples of this include division by zero, nil-pointer dereference and class cast errors. The standard solution to this is to lift the semantics and include these errors into the valid results of the execution, often in the form of exceptions, and to have the type system rule out all errors not modeled in the semantics, typically in addition to tracking what errors a program may result in.

For a standard type system this solution is adequate; the types of programs are not affected in any other way than the addition of a set of possible exceptions. In particular, any inaccuracies in the set of possible exceptions are unproblematic to the type system itself (albeit inconvenient to the programmer), and, thus, in standard programming

languages not much effort is made to rule out syntactically present but semantically impossible exceptions.

For type based program analyses, however, the situation is different. Not only are we not able to change the semantics to fit the capabilities of the type system, since we are, in effect, retrofitting a type system onto an existing language, but for some analyses — notably, for *information flow security* type systems [15] — inaccuracies propagate from e.g. the exception types via *implicit flows* to the other types lowering the precision of the type system and possibly rendering more semantically secure programs to be classified as insecure. Consider the following example:

```
try c1; c2; ... catch (Exception e) ch
```

If the command c_1 may fail this affects whether the succeeding commands $c_2, \ldots$ are run or not, and thus any side effects — e.g. output on a public network — will encode information about the data manipulated by c_1. If this information must be protected, this put serious limits on the succeeding commands $c_2, \ldots$ and on the exception handler ch.

This is problematic, since dynamic error handling introduces many possible branches — every partial instruction becomes a possible branch to the error handler *if it cannot be guaranteed not to crash*, and, thus, a source of implicit flows. Hence, from a practical standpoint, there is a need to increase the accuracy of type based information flow analyses as demonstrated by some recent attempts [2, 12, 1]. Noting that the majority of the information flow analyses are formulated in terms of type systems, we focus on how to strengthen a type system with additional information to increase its accuracy.

Even though our main motivation for this work comes from information flow type systems, we investigate the problem in terms of a standard type system; this both generalizes the method and simplifies the presentation. All our results are immediately applicable to information flow type systems.

We see two major different methods of solving the problem of strengthening type systems: 1) by *integration*, and 2) by *parameterization*. Briefly, 1) relies on extending the type system to compute the additional needed information, and 2) relies on using information about the programs' execution provided by other analyses. Integration is problematic for a number of reasons: 1) it obscures the original intention of the type system especially if multiple additional analyses are added, 2) it makes use of already available analyses difficult, since they have to be rephrased as type systems, and 3) it is non-modular: changing the supporting analysis implies changing the entire type system.

Contribution We present a modular approach for parameterizing type systems with information about the program execution; the method is modular not only at the type system level, but also at the proof level.

The novelty of the approach lies not in the idea of parameterizing information in itself, rather, the novelty is the setting — the parameterization of *type systems* with information from *abstract analyses* — together with the identification of a general, widely applicable format for information passing and inspection, which allows for modularity with only small modifications to the type system and its correctness proof, and no modifications to the abstract analyses. This modularity makes instancing parameterized type systems with the results of different abstract analyses relatively cheap, which can be

leveraged to create staged type systems, where increasingly precise analyses are chosen based on previous typing errors.

Finally, we exemplify the use of the method in terms of a parameterized type system for a small imperative language with arrays, and explore how the parameterization can be used to rule out nil-pointer exceptions, and exceptions stemming from array indices outside the bounds of the corresponding arrays.

Outline Section 2 presents a small imperative language with arrays, used to explain the method more concretely. Section 3 presents the parameterization: the abstract environment maps, the plugins properties and the plugins, and describes the process of parameterizing a type system. Section 4 is a concrete example of applying the method to get a parameterized type system of the language of Section 2. Section 5 discusses related work, and finally Section 6 concludes and discusses future work.

2 Language

To be concrete we use a small imperative language with arrays to illustrate our method.

Syntax The language is a standard while language with arrays. For simplicity we consider all binary operators to be total; the same techniques described to handling the partiality of array indexing apply to partial operators. The syntax of the language is found in Table 1, where the allocation type $\tau[i]$ indicates that an array of size i with elements of type τ should be allocated; the primitive types ranged over by τ are defined in Section 4.1 below.

Expressions $e ::= nil \mid i \mid x \mid e \star e \mid x[e] \mid len(x)$
Commands $c ::= x := e \mid x[e] := x \mid \mathit{if}\ e\ c\ c \mid \mathit{while}\ e\ c \mid c; c \mid x := new(\tau[i]) \mid \mathit{skip}$

Table 1. Syntax

Semantics The semantics of the expressions is given in terms of a big step semantics with transitions of the form $\langle E, e\rangle \Downarrow v_\perp$, where $v_\perp$ ranges over error lifted values v ($\perp$ indicates errors), and E ranges over the set of environments Env, i.e. pairs (s, h) of a store, and a heap. The values consist of the integers i and the pointers p. The arrays a are pairs (i, d) of the size of the array and a map from integers to values with a continuous domain starting from 0. Formally, d ranges over $\bigcup_{n\in\mathbb{N}}\{[0 \mapsto v_1, \ldots, n \mapsto v_n]\}$. The stores s are maps from variables x to values, and the heaps h are maps from pointers to arrays.

In the definition of the semantics, if $a = (i_1, d)$ then let $a(i_2)$ denote $d(i_2)$. Further, for $E = (s, h)$, let $E(x)$ denote $s(x)$, $E[x \mapsto v]$ denote $(s[x \mapsto v], h)$, $E(p)$ denote $h(p)$, and similarly for other operations on environments including variables or pointers.

The semantics of commands is given in terms of a small step semantics between configurations C with transitions of the form $\langle E, c\rangle \to C$, where C is either one of the terminal configurations $\perp_E$ and $\langle E, \mathit{skip}\rangle$ indicating abnormal and normal termination

in the environment E, respectively, or a non-terminal configuration $\langle E, c\rangle$ where $c \neq \mathit{skip}$. A selection of the semantic rules for expressions and commands are presented in Table 2; the omitted rules are found in the extended version of this paper [9].

$$\frac{E(x) = p \quad E(p) = (i, d)}{\langle E, \mathit{len}(x)\rangle \Downarrow i} \qquad \frac{E(x) = \mathit{nil}}{\langle E, \mathit{len}(x)\rangle \Downarrow \bot} \qquad \frac{E(x_1) = \mathit{nil}}{\langle E, x_1[i] := x_2\rangle \to \bot_E}$$

$$\frac{E(x_1) = p \quad E(p) = (i_2, d) \quad E(x_2) = v \quad i_1 \notin [0..(i_2 - 1)]}{\langle E, x_1[i_1] := x_2\rangle \to \bot_E}$$

$$\frac{E(x_1) = p \quad E(p) = (i_2, d) \quad E(x_2) = v \quad i_1 \in [0..(i_2 - 1)]}{\langle E, x_1[i_1] := x_2\rangle \to \langle E[p \mapsto (i_2, d[i_1 \mapsto v])], \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{while}\ e\ c\rangle \to \langle E, \mathit{if}\ e\ (c; \mathit{while}\ e\ c)\ \mathit{skip}\rangle}$$

$$\frac{\langle E, e\rangle \Downarrow v}{\langle E, R[e]\rangle \to \langle E, R[v]\rangle} \qquad \frac{\langle E, e\rangle \Downarrow \bot}{\langle E, R[e]\rangle \to \bot_E} \qquad \frac{E(x) = p \quad E(p) = (i_1, d) \quad \langle E, e\rangle \Downarrow i_2 \quad i_2 \notin [0..i_1 - 1]}{\langle E, x[e]\rangle \Downarrow \bot}$$

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, R[c_1]\rangle \to \langle E_2, R[c_2]\rangle} \qquad \frac{\langle E, c\rangle \to \bot_E}{\langle E, R[c]\rangle \to \bot_E}$$

Table 2. Selected Semantic Rules for Expressions and Commands

As is common for small step semantics we use evaluation contexts R.

$$R ::= \cdot \mid x := R \mid x[R] := x \mid \mathit{if}\ R\ c\ c \mid R; c$$

The accompanying standard reduction rules allow for leftmost reduction of sequences, error propagation and reduction of expressions inside commands.

3 Parameterization

With this we are ready to detail the method of parameterization. First, let us recapture our goal: we want to describe a modular way of parameterizing a type system with information about the programs' execution in such a way that a modular correctness proof can be formed for the resulting system, with the property that an instantiated system satisfies a correspondingly instantiated correctness proof.

To achieve this, we define a general format of parameterized information and a general method to access this information. Using the ideas of abstract interpretation, we let the parameterized information be a map from program points to abstract environments, intuitively representing the set of environments that can reach each program point. Such a map is semantically sound — a *solution* in our terminology — w.r.t. a set of initial concrete environments and a program, if every possible execution trace the initial environments can give rise to is modeled by the map.

For modularity we do not want to assume anything about the structure of the abstract environments, but treat them as completely opaque. Noting that each type system uses a finite number of forms of questions, we parameterize the type system not only over the abstract environment map, but also over a set of *plugins* — sound approximations of the semantic properties of the questions used by the type system.

Labeled Commands Following the elegant approach of Sands and Hunt [13] we extend the command language with label annotations, which allow for a particularly direct way of recording the environments that enter and leave the labeled commands. Let l range over labels drawn from the set of labels $\mathcal{L}$. A command c can be annotated with an entry label $(c)^l$, an exit label $(c)_l$, or both.

We extend the reduction contexts with $(R)_l$, which allows for reduction under exit labels, and the semantics with the following transitions.

$$\frac{}{\langle E, (c)^l\rangle \rightarrow \langle E, c\rangle} \qquad \frac{}{\langle E, (\mathit{skip})_l\rangle \rightarrow \langle E, \mathit{skip}\rangle}$$

The idea is that a transition of the form $\langle E, (c)^l\rangle \rightarrow \langle E, c\rangle$ leaves a marker in the execution sequence that the command labeled with the entry label l was executed in E, and a transition of the form $\langle E, (\mathit{skip})_l\rangle \rightarrow \langle E, \mathit{skip}\rangle$ indicates that the environment E was produced by the command labeled with the exit label l, which is why allowing reduction under exit labels but not under entry labels is important.

3.1 Abstract Environment Maps

Using the ideas of abstract interpretation [4, 5], let $\mathbb{Env}$ be the set of abstract environments ranged over by $\mathbb{E}$, equipped with a concretization function $\gamma : \mathbb{Env} \rightarrow \mathcal{P}(Env)$, and let an abstract environment map $\mathbb{M} : \mathcal{L} \rightarrow \mathbb{Env}$ be a map from program points to abstract environments, associating each program point with an abstract environment representing all concrete environments that may reach the program point.

We define two soundness properties for abstract environment maps that relate the maps to the execution of a program when started in environments drawn from a set of initial environments $\mathcal{C}$.

An abstract environment map $\mathbb{M}$ is an *entry solution* written $\mathit{entrysol}^{E_1}_{c_1}(\mathbb{M})$ w.r.t. an initial concrete environment E_1, and a program c_1 if all $\langle E_2, (c_2)^l\rangle \rightarrow \langle E_2, c_2\rangle$ transitions in the trace originating in $\langle E_1, c_1\rangle$ are captured by $\mathbb{M}$. The notion of *exit solution* written $\mathit{exitsol}^{E_1}_{c}(\mathbb{M})$ is defined similarly but w.r.t. all transitions of the form $\langle E_2, (\mathit{skip})_l\rangle \rightarrow \langle E_2, \mathit{skip}\rangle$.

$$\begin{aligned}\mathit{entrysol}^{E_1}_{c_1}(\mathbb{M}) &\equiv \forall E_2, c_2, l\,.\,\langle E_1, c_1\rangle \rightarrow^* \langle E_2, R[(c_2)^l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l))\\ \mathit{exitsol}^{E_1}_{c}(\mathbb{M}) &\equiv \forall E_2, l\,.\,\langle E_1, c\rangle \rightarrow^* \langle E_2, R[(\mathit{skip})_l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l))\end{aligned}$$

The definitions are lifted to sets of initial environments $\mathcal{C}$ in the obvious way.

Both the entry and exit solution properties are preserved under execution as defined below.

Lemma 1 (Preservation of Entry and Exit Solutions under Execution). *In the following, let $\mathcal{C}_1$ be the set of initial concrete environments and $\mathcal{C}_2$ the set of environments that reach c_2, i.e. $\mathcal{C}_2 = \{E_2 \mid E_1 \in \mathcal{C}_1,\ \langle E_1, c_1\rangle \rightarrow \langle E_2, c_2\rangle\}$.*

$$\mathit{entrysol}^{\mathcal{C}_1}_{c_1}(\mathbb{M}) \Longrightarrow \mathit{entrysol}^{\mathcal{C}_2}_{c_2}(\mathbb{M}) \textit{ and } \mathit{exitsol}^{\mathcal{C}_1}_{c_1}(\mathbb{M}) \Longrightarrow \mathit{exitsol}^{\mathcal{C}_2}_{c_2}(\mathbb{M})$$

These properties immediately extend to any finite sequence of execution steps by inductions over the length of the sequence.

Further, solutions can freely be paired to form new solutions similarly to the independent attribute method for abstract interpretation [5]. This is important since it shows that no generality is lost by parameterizing a type system over only one abstract environment map.

3.2 Plugins

To the parameterized type systems, the structure of the abstract environments is opaque and cannot be accessed directly. This allows for the decoupling of the parameterized type system and the external analysis computing the abstract environments. However, the parameterized type systems need a way to extract the desired information. To this end we introduce the concept of *plugins*. Intuitively, a plugin provides information about a specific property of an environment; for instance, a nil-pointer plugin provides information about which parts of the environment are nil.

The plugins are defined to be sound approximations of *plugin properties*, defined as families of relations on expressions.

Plugin Properties Let R be an n-ary relation on values; R induces a *plugin property*, written $R^\diamond$, which is a family of n-ary relations on expressions indexed by environments in the following way.

$$(e_1, \ldots, e_n) \in R^\diamond_E \iff \langle E, e_1\rangle \Downarrow v_1 \wedge \ldots \wedge \langle E, e_n\rangle \Downarrow v_n \implies (v_1, \ldots, v_n) \in R$$

We can use the expression language to define semantic properties about environments, since the expression language is simple, in particular, since it does not contain iteration, and is free from side effects. A major advantage of the approach is that it allows for a relatively simple treatment of expressions in programs.

The choice of using the expression language as the plugin language is merely out of convenience — languages with richer expression language would mandate a separate language for the plugins and treat the exceptions similarly to the statement, i.e. extend the labeling and the solutions to the expressions. In our case, however, a separate plugin language would be identical to the expressions.

Example 1 (Non-nil and Less-than Plugin Properties). The non-nil plugin property $nn^\diamond$ can be defined by a family of predicates indexed over concrete environments induced by the value property *nn* defined such that $nn(v)$ holds only if the value v is not equal to nil. Similarly, the less-than plugin property $lt^\diamond$ can be defined by *lt* such that $lt(v_1, v_2)$ holds only if the value v_1 is less than the value v_2. □

Plugins A plugin is a family of relations on expressions indexed by abstract environments. Given a plugin property $R^\diamond$ we define *plugins*, $R^\sharp$ as follows.

$$(e_1, \ldots, e_n) \in R^\sharp_{\mathbb{E}} \implies \forall E \in \gamma(\mathbb{E}).\ (e_1, \ldots, e_n) \in R^\diamond_E$$

It is important to note that for each plugin property there are many possible plugins, since the above formulation allows for approximative plugins. This means that regardless of the abstract environment, and the decidability of the plugin property $R^\diamond$, there exist decidable plugins, which guarantees the possibility of preservation of decidability for parameterized type systems.

Example 2 (Use of Plugins). Assume a type system computing a set of possibly thrown exceptions. When typing, for example, the array length operator $len(x)$ we are interested in the plugin property given by the non-nil predicate *nn*. Let $\mathbb{E}$ be a sound representation of all environments reaching $len(x)$. Given $nn^\sharp_{\mathbb{E}}(x)$, we know that x will not be nil in any of the concrete environments represented by $\mathbb{E}$, and, since $\mathbb{E}$ is a sound representation of all environments that can reach the array length operator, we know that a nil-pointer exception will not be thrown. □

Despite the relative simplicity of the plugin format it is surprisingly powerful; in addition to the obvious information, such as is x ever nil, it turns out that plugins can be used to explore the structure of the heap as we show in [10] where we use the parameterization to provide flow sensitive heap types.

3.3 Overview of a Parameterized Type System

Assume an arbitrary flow insensitive type system[1] of the form $\Gamma \vdash_A c$ expressing that c is well-typed in the type signature Γ, under the additional assumption A. We let the exact forms of Γ and A be abstract; however, typical examples are that Γ is a store type, and, for information flow type systems, that A is the security level of the context, known as the pc [15].

The first step in parameterizing the type system is to identify the plugin properties $R^\diamond_1, \ldots, R^\diamond_m$ that are to be used in the parameterized type rules. For instance the non-nil plugin property can be used to increase the precision of the type rule for the array length operator as discussed in Example 2 above, cf. the corresponding type rules in Section 4 below. Each type rule is then parameterized with an abstract environment map $\mathbb{M}$, and a number of plugins $R^\sharp_1, \ldots R^\sharp_m$, one for each of the plugin properties, forming a parameterized system of the following form.

$$\Gamma \vdash^{\mathbb{M}, R^\sharp_1, \ldots, R^\sharp_m}_{A} c$$

A typical correctness argument for type systems is *preservation* [14], i.e. the preservation of a type induced invariant, well-formedness, (see Section 4.3 below) under execution. Well-formedness defines when an environment conforms to an environment type, e.g. that all variables of integer type contain integers. Let $wf_\Gamma(E)$ denote that E is well-formed in the environment type Γ; a typical preservation statement has the following form:

$$\Gamma \vdash_A c \Longrightarrow wf_\Gamma(E_1) \wedge \langle E_1, c\rangle \rightarrow E_2 \Longrightarrow wf_\Gamma(E_2)$$

More generally, a class of correctness arguments for type systems have the form of preservation of an arbitrary type indexed relation $\mathcal{R}_\Gamma$ under execution:

$$\begin{aligned}\Gamma \vdash_A c \Longrightarrow\ & \mathcal{R}_\Gamma(E_{11}, \ldots, E_{1n}) \wedge \\ & \langle E_{11}, c\rangle \rightarrow E_{21} \wedge \cdots \wedge \langle E_{1n}, c\rangle \rightarrow E_{2n} \Longrightarrow \mathcal{R}_\Gamma(E_{21}, \ldots, E_{2n})\end{aligned}$$

[1] Our method works equally well for flow sensitive type systems, but for brevity of explanation this section is done in terms of a flow insensitive system.

This generalization is needed to capture invariants that are not *safety properties*, for instance noninterference or live variable analysis.

For *conservative* parameterizations, i.e. where we add type rules with increased precision, the proofs of correctness are essentially identical to the old proofs, where certain execution cases have been ruled out using the semantic interpretation of the plugins. To see this consider that a typical proof of the above lemma proceeds with a case analysis on the possible ways c can execute in the different environments E_{11} to E_{1n} and proves the property for each case. See the proof of Theorem 1 in Section 4.3 for an example of this. The correctness statement for the parameterized types system becomes:

$$entrysol_c^{\mathcal{C}}(\mathbb{M}) \wedge E_{11} \in \mathcal{C} \wedge \cdots \wedge E_{1n} \in \mathcal{C} \wedge \Gamma \vdash_A^{\mathbb{M}, R_1^\sharp, \ldots, R_m^\sharp} c \Longrightarrow$$
$$\mathcal{R}_\Gamma(E_{11}, \ldots, E_{1n}) \wedge \langle E_{11}, c\rangle \rightarrow E_{21} \wedge \cdots \wedge \langle E_{1n}, c\rangle \rightarrow E_{2n} \Longrightarrow \mathcal{R}_\Gamma(E_{21}, \ldots, E_{2n})$$

The interpretation of this statement is that execution started in any of the environments in the set of possible initial environments is $\mathcal{R}_\Gamma$-preserving, i.e. it narrows the validity of the original lemma to the set of initial environments.

Proof of Correctness It is important to note that we do not need to redo any parts of the correctness proof when instantiating a parameterized type system.

The assumption that the extracted abstract environment maps are sound for the program and set of initial environments under consideration, i.e. that they are solutions, is established once per family of external analysis. This is established by, e.g., formulating the family as a family of abstract interpretations and proving that all environment maps extracted from an abstract analysis belonging to the family are sound for the program and set of initial environments that the analysis was started with.

What is left per instantiation is to show that the used plugins are valid. In most cases, this is trivial since the structure of the abstract environments have been chosen with this in mind. Furthermore, many type systems can be improved with similar information; thus, it should be possible to build a library with plugins for different plugin properties that can be used when instantiating implementations of parameterized type systems. This is important because it shows that creating new correct instantiations is a comparatively cheap operation, which leads to interesting implementation possibilities.

4 A Parameterized Type System

In this section we exemplify the ideas described in the previous section by presenting a parameterized type system for the language introduced in Section 2. The type system improves over the typical type system for such a language by using the parameterized information to rule out exceptions that cannot occur.

A larger example of a parameterized type system, showing how plugins can be used to perform structural weakening and strong updates for a flow-sensitive type system, can be found in [10].

4.1 Type Language

The primitive types ranged over by τ are the type of integers int, and array types, $\tau[]$, indicating an array with elements of type τ. The store types ranged over by Σ are maps from variables to primitive types. The exception types ranged over by ξ are $\bot_\Sigma$, indicating the possibility that an exception is thrown, and $\top$, indicating that no exception is thrown. This is a simplification from typical models of exceptions, where multiple types are used to indicate the reason for the exception. However, for the purpose of exemplifying the parameterization this model suffices; the results are easily extended to a richer model. In addition we use a standard subtype relation $<:$(omitted for space reasons) with invariant array types.

4.2 Type Rules

The judgments for expressions, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$, is read as the expression e is well-typed w.r.t. the abstract environment $\mathbb{E}$, the non-nil plugin $nn^\sharp$, and the less-than plugin $lt^\sharp$, in the environment type Σ, with return type τ possibly resulting in exceptions as indicated by ξ. The type system for commands is flow-sensitive; the judgment, $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi$ is read as the command c is well-typed w.r.t. the abstract environment map $\mathbb{M}$, the non-nil plugin $nn^\sharp$, and the less-than plugin $lt^\sharp$, in the environment type Σ_1 resulting in the environment type Σ_2, possibly resulting in an exception as indicated by ξ. The relevant type rules for expressions and commands are found in Table 3 where we use $\vdash^\dagger$ as short notation for $\vdash^{\mathbb{M},nn^\sharp,lt^\sharp}$ and $\vdash^\ddagger$ as short notation for $\vdash^{\mathbb{E},nn^\sharp,lt^\sharp}$; the omitted rules are found in the extended version of this paper [9].

$$\frac{\Sigma \vdash^{\mathbb{M}(l),nn^\sharp,lt^\sharp} e : int, \xi \quad \Sigma(x_1) = \tau_1[] \quad \Sigma(x_2) = \tau_2 \quad \tau_2 <: \tau_1 \quad \neg(nn^\sharp_{\mathbb{M}(l)}(x_1) \wedge -1\ lt^\sharp_{\mathbb{M}(l)}\ e \wedge e\ lt^\sharp_{\mathbb{M}(l)}\ len(x_1))}{\Sigma \vdash^\dagger (x_1[e] := x_2)^l \Rightarrow \Sigma, \bot_\Sigma}$$

$$\frac{\Sigma \vdash^{\mathbb{M}(l),nn^\sharp,lt^\sharp} e : int, \xi \quad \Sigma(x_1) = \tau_1[] \quad \Sigma(x_2) = \tau_2 \quad \tau_2 <: \tau_1 \quad nn^\sharp_{\mathbb{M}(l)}(x_1) \quad -1\ lt^\sharp_{\mathbb{M}(l)}\ e \quad e\ lt^\sharp_{\mathbb{M}(l)}\ len(x_1)}{\Sigma \vdash^\dagger (x_1[e] := x_2)^l \Rightarrow \Sigma, \xi}$$

$$\frac{\Sigma(x) = \tau[] \quad \neg nn^\sharp_{\mathbb{E}}(x)}{\Sigma \vdash^\ddagger len(x) : int, \bot_\Sigma} \qquad \frac{\Sigma(x) = \tau[] \quad nn^\sharp_{\mathbb{E}}(x)}{\Sigma \vdash^\ddagger len(x) : int, \top}$$

Table 3. Selected Type Rules for Expressions and Commands

Apart from the parts related to the parameterization, the expression and command type rules are entirely standard. With respect to the parameterization specifics, the type rules for array size, and array indexing make use of the parameterized information and occur in two forms: one that is able to exclude the possibility of exceptions, and one that is not.

For the array size operator it suffices to rule out that the variable x ever contains nil to rule out the possibility of exceptions, for array indexing (for both the expression and the command) we must demand that the index is greater or equal to zero, and that the

index is smaller than the size of the array in addition to the demand that the variable is non-nil. For an example detailing the type derivation of a small program with different parameterized information see the example in the extended version of this paper [9].

4.3 Correctness

With this we are ready to formulate correctness for the parameterized type system. As is standard we split the correctness argument into two theorems, *progress* — intuitively, that well-typed commands and expressions are able to execute in all environments that conform to the entry environment type of the command or expression — and *preservation* — intuitively, that the result of running the command or expression conforms to the exit type of the same. In contrast to the preservation proof, the progress proof is independent of the parameterized information. For space reasons we omit the progress proof.

Well-formedness The well-formedness relation in Table 4 defines when a context is well-formed w.r.t. a type. It is the extension of a standard well-formedness relation to exception types. Most of the standard well-formedness relation has been omitted for space reasons and is found in the extended version of this paper [9]. In short, a value

$$\frac{\delta \vdash v : \tau}{\delta \vdash v : \tau, \xi} \qquad \frac{}{\delta \vdash \bot : \tau, \bot_\Sigma}$$

$$\frac{\delta \vdash E : \Sigma_2}{\delta \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} \bot_E : \Sigma_1, \bot_{\Sigma_2}} \qquad \frac{\delta \vdash E : \Sigma}{\delta \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} E : \Sigma, \xi}$$

$$\frac{\Sigma_1 \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} c \Rightarrow \Sigma_2, \xi \quad \delta \vdash E : \Sigma_1}{\delta \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} \langle E, c \rangle : \Sigma_2, \xi} \qquad \frac{\forall i \in dom(a) \,.\, \delta \vdash a[i] : \tau}{\delta \vdash a : \tau[]}$$

Table 4. Well-formedness

is well-formed w.r.t. any exception type, whereas an error is only well-formed w.r.t. an exception type that indicates the possibility of the error, and similarly for well-formed environments, with the addition of the demand that the exception environment is well-formed in the exception environment type. A configuration is well-formed in the type Σ_2, ξ if there exists an environment type Σ_1 in which the environment E is well-formed such that the command is well-typed with Σ_1 as entry type and the Σ_2, ξ as exit type.

Preservation of Types of Expressions and Commands Preservation of types of expressions expresses that well-typed expressions preserve well-formedness under execution, i.e. for an expression e s.t. $\Sigma \vdash^{\mathbb{E}, nn^\sharp, lt^\sharp} e : \tau, \xi$ running e in Σ-well-formed environments that are modeled by the abstract environment $\mathbb{E}$ will result in τ, ξ-well-formed values.

Theorem 1 (Preservation of Types of Expressions).

$$\Sigma \vdash^{\mathbb{E}, nn^\sharp, lt^\sharp} e : \tau, \xi \implies \forall E \in \gamma(\mathbb{E}) \,.\, \delta \vdash E : \Sigma \wedge \langle E, e \rangle \Downarrow v \implies \delta \vdash v : \tau, \xi$$

Proof. By induction on the derivation of $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$. Intuitively, in each case, the proof proceeds by an inversion of $\langle E, e\rangle \Downarrow v$, which results in a number of subcases — one for each semantic rule for the expression, including the ones resulting in exceptions. However, in the cases where the type system can rule out exception it contains enough information about the execution from the use of the plugins on the abstract environment to disprove the possibility of an exception.

We exemplify the difference between a standard proof and a parameterized proof by proving the correctness for the array indexing cases, corresponding to the two type rules for array indexing — for space reasons, in the cases the antecedents of the expression type rules and semantics rules are subsets of their command counterparts the expression rules have been omitted in this version of the paper, and we refer the reader to the command type rules in order to follow the proof below.

Assume $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$, (2) $E \in \gamma(\mathbb{E})$, (3) $\delta \vdash E : \Sigma$ and (4) $\langle E, e\rangle \Downarrow v$. We must show $\delta \vdash v : \tau, \xi$.

array indexing with exceptions In this case the last applied type rule in the derivation is the rule that cannot rule out exceptions, which gives (5) $\Sigma(x) = \tau[]$, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e' : int, \xi'$, $\neg(nn^\sharp_{\mathbb{E}}(x) \wedge -1\ lt^\sharp_{\mathbb{E}}\ e' \wedge e'\ lt^\sharp_{\mathbb{E}}\ len(x))$, $\xi = \bot_\Sigma$ and that $e = x[e']$. Inversion of (4) gives us the following four cases.

- **1) nil-pointer exception** This case gives $v = \bot$ from which the result $\delta \vdash \bot : \tau, \bot_\Sigma$ is immediate.
- **2)** e **leads to an exception** Same as the case above.
- **3) index out of bounds** Same as the case above.
- **4) successful execution** Let $E = (s, h)$; this case gives (6) $s(x) = p$, $h(p) = (i_1, d)$, $\langle E, e'\rangle \Downarrow i_2$, (7) $i_2 \in [0..(i_1 - 1)]$ and $v = d(i_2)$. From $(3, 5, 6)$ we get $\delta \vdash p : \tau[]$, which in turn gives $\delta(p) <: \tau[]$, which means (8) $\delta(p) = \tau[]$, since array subtyping is invariant. Further, (3) and (8) give $\delta \vdash h(p) : \tau[]$, which gives $\forall i \in dom((i_1, d))\ .\ \delta \vdash (i_1, d)(i) : \tau$. Thus, (7) gives us that $i_2 \in dom((i_1, d))$, from which we get the result $\delta \vdash d(i_2) : \tau$.

array indexing without exceptions In this case the last applied type rule in the derivation is the rule that rules out exceptions, which gives $\Sigma(x) = \tau[]$, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e' : int, \xi$, (5) $nn^\sharp_{\mathbb{E}}(x)$, (6) $-1\ lt^\sharp_{\mathbb{E}}\ e'$, (7) $e'\ lt^\sharp_{\mathbb{E}}\ len(x)$, and that $e = x[e']$. Again, inversion of (4) gives us the following four cases.

- **1) nil-pointer exception** This case gives (8) $s(x) = nil$. (1) and (5) give $\forall E \in \gamma(\mathbb{E}).\ x \in nn^\diamond_E$, which together with (2) gives (9) $\langle E, x\rangle \Downarrow nil \Longrightarrow nil \in nn$. (8) gives $\langle E, x\rangle \Downarrow nil$, which together with (9) gives $nil \in nn$ which is a contradiction.
- **2)** e **leads to an exception** This case gives $\langle E, e'\rangle \Downarrow \bot$ which together with the induction hypothesis gives $\xi = \bot_\Sigma$ from which the result is immediate.
- **3) index out of bounds** This case gives $s(x) = p$, $h(p) = (i_1, d)$, $\langle E, e'\rangle \Downarrow i_2$ and (8) $i_2 \notin [0..(i_1 - 1)]$. In a way similar to 1) above, we use (6) to disprove that i_2 is less than 0 and (7) to disprove that i_2 is greater than or equal to i_1. Together this contradicts (8) and we have reached a contradiction.
- **4) successful execution** This case is proven in the same way as case 4) in array indexing with exceptions.

Thus, as the proof of preservation of types for array indexing shows, we achieve higher precision in the exception type by using the parameterized information to disprove some cases as described in Section 3.3. As discussed, the proof for the parameterized type system is essentially identical to the original proof where there is no parameterized information, with the difference that two cases are disproved.

Given the well-formedness formulation for configurations above, preservation of types of commands can be formulated in the same way as preservation of types of expressions.

Theorem 2 (Preservation of Types of Commands).

$$\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi \wedge entrysol_c^{\mathcal{C}}(\mathbb{M}) \Longrightarrow$$
$$\forall E \in \mathcal{C} \,.\, \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c\rangle \rightarrow C \Longrightarrow \exists \delta_2 \,.\, \delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} C : \Sigma_2, \xi$$

Proof. For space reasons the proof is found in the extended version of this paper [9].

Top-level Correctness of Commands Let $\langle E, c\rangle \rightarrow^n C$ be the obvious lifting of the small step evaluation to evaluation of n consecutive steps. With this we are ready to formulate the top-level correctness of commands, that well-typed commands terminate in a well-formed environment or result in well-formed configurations regardless of the number of execution steps. For convenience we let T range over terminal configurations.

Theorem 3 (Top-level Correctness of Commands).

$$\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_1 \Rightarrow \Sigma_2, \xi \wedge entrysol_{c_1}^{\mathcal{C}}(\mathbb{M}) \Longrightarrow \forall E_1 \in \mathcal{C} \,.\, \delta_1 \vdash E_1 : \Sigma_1 \Longrightarrow$$
$$\forall n. \, (\exists n' \leq n, T, \delta_2. \, \langle E_1, c_1\rangle \rightarrow^{n'} T \wedge \delta_2 \vdash T : \Sigma_2, \xi) \vee$$
$$(\exists E_2, c_2, \delta_2. \, \langle E_1, c_1\rangle \rightarrow^n \langle E_2, c_2\rangle \wedge \delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} \langle E_2, c_2\rangle : \Sigma_2, \xi)$$

Proof. For space reasons the proof is found in an extended version of this paper [9].

5 Related Work

The method presented in this paper combines an analysis, formulated as a type system, with a number of external analyses, computing information useful to the type system, by parameterizing the type system over the computed information.

Similar in spirit is the work by Foster, Fähndrich and Aiken [8] in which they present a framework for augmenting existing type systems with type qualifiers, e.g. `const` and `nonnil`. Our work differs from theirs in that they provide a framework to compute the qualifiers, rather than making use of them.

In [3] Chin, Markstrum and Millstein investigate a method for supporting user-defined semantic type qualifiers that are closely related to unary plugins. As above, their work is aimed at computing an analysis result, rather than modularly making use of it. In addition to reason about soundness they propose a method to automatically verify the soundness of the extension using an automatic theorem prover.

Among the type systems making use of additional information are type systems that eliminate array bound checks, e.g. [16], using a decidable formulation of dependent types. It should be pointed out that even though the type checking is decidable the inference is not; nothing in our approach rules out inference. In [12] Hedin and Sands use a simplistic type based inference of nil-pointers needed to allow the use of non-secret fields in objects pointed to by pointers with secret pointer values. We believe that the clarity, correctness proof and power of their system could benefit greatly by being reformulated in our framework.

In [6] Crary and Weirich present a type system for resource bound verification, e.g. memory usage and execution time. Their type system goes beyond the capacity of the plugins framework — time and memory usage are not values in a standard semantics. It could potentially be interesting to see to what extent the plugins model can be modified to encompass such extensions.

While this work suggests resolving type errors by using more and more elaborate parameterized analyses, Flanagan [7] suggests pushing checks that cannot be statically resolved to runtime checks, cf. type cast checks in Java. For many uses of the plugins framework, uniting the two approaches could prove beneficial — if the program cannot be statically proven correct using a different external analysis, Flanagan's method could be applied to insert a dynamic check.

With respect to other work on combining static analyses, if the analyses we want to combine are formulated as abstract interpretations, a number of techniques from the large body of work on abstract interpretation [4, 5, 11] becomes applicable. An example of such a combination is the reduced product method. Similar to our method, the combination can be done in a systematic way and correctness of the resulting analysis follows from correctness of the combined analyses.

An advantage of the abstract interpretation framework is that for partially overlapping analyses and a combination like the reduced product, the analyses will benefit from each other. Each analysis can make use of the information computed by the other analyses, which stands in contrast to our method where the external analyses cannot make direct use of the derivation of the parameterized type system.

However, an obvious restriction of the abstract interpretation framework is that all analyses must be formulated as abstract interpretations, which is not always the case. Reformulating, for example, a type based analysis into an abstract interpretation is not always easily done nor desirable, as for example indicated by the field of security where the analyses tend to be type based [15]. Our approach does not have that restriction. A type system can be combined with any external analyses that compute valid solutions. If the external analyses are formulated as abstract interpretations our method can be combined with the abstract interpretation framework to make use of, for example, reduced products.

6 Conclusions and Future Work

We have presented a method for parameterizing program analyses for imperative small step languages with information about the programs' execution. The appeal of the method compared to approaches where additional information about the programs' ex-

ecution is provided by extending the type system with capabilities of computing the additional information, i.e. fusing the type system with another analysis, lies in that:

- The parameterization does not impose heavy changes to the type system. The rules remain relatively close to the original rules; only the use of the additional information is added to the rules where the information is used — other rules remain essentially unaffected. Comparatively, fusing an analysis modifies all rules to compute the information, in addition to the uses of the information in certain rules.
- The parameterization gives the possibility of changing the parameterized analysis with relative ease — proofs for the family of analyses[2], and decision procedures with corresponding soundness proofs have to be done. Comparatively, changing the analysis for a fused type system means creating a new fused type system and correctness proof from scratch.

The method is based on the identification of a generic format for information exchange between the program analysis and the parameterized results, together with methods — the plugins — for asking specific questions about the each program parts execution environment.

To exemplify the method we have given an overview of the steps involved in parameterizing an existing type system, including the changes to the type system itself, but also the changes to the correctness proof of the type system. A corner stone in this work is the attempt to make the correctness proof a natural part of the parameterization process so that the proof burden for each parameterization is relatively low.

A drawback is that the resulting system may no longer be compositional; e.g. a compositional type system becomes non-compositional if the parameterized information is not compositional. Another restriction is that the parameterization is one-way only; there is no back propagation of type information that could have been used by the parameterized analysis.

Future Work The motivation for this work grew out of a perceived need to increase the precision of type based analyses of secure information flow. For this reason a natural continuation of this work is to apply the method to an information flow type system.

In addition to this, an implementation of the parameterized type system of this paper would be valuable to asses the practicality of the approach. Of particular interest would be to implement a staged type system, where the reason for a type failure is analyzed and given as feedback to the next stage. The benefit of doing this is apparent in cases where the abstract environment map is a combination of the result of a number of external analyses. One way to view a set of increasingly precise external analyses is as a matrix with one dimension for each type of analysis and plugin property. In the general setting where a parameterized type system uses multiple external analyses the external analyses build up a multi-dimensional matrix where each point corresponds to a particular instantiation of the type system.

[2] The proof only has to be done once for each family, and typically includes a way of converting the analysis information provided by the family to the format of the parameterization.

Acknowledgements This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 91–102, New York, NY, USA, 2006. ACM Press.
2. Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.
3. Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. *SIGPLAN Not.*, 40(6):85–95, 2005.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
5. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
6. Karl Crary and Stephnie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM.
7. Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM.
8. Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5):192–203, 1999.
9. Tobias Gedell and Daniel Hedin. Abstract interpretation plugins for type systems. Technical Report 2008:10, Computing Science Department, Chalmers.
10. Tobias Gedell and Daniel Hedin. Plugins for structural weakening and strong updates. *Unpublished*.
11. Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM Press.
12. David Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
13. Sebastian Hunt and David Sands. Just forget it – the semantics and enforcement of information erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, number 4960 in LNCS, pages 239–253. Springer Verlag, 2008.
14. Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.
15. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
16. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Not.*, 33(5):249–257, 1998.