Securing Perl with Type Inference

Gary Jackson <bargle@umiacs.umd.edu>

May 20, 2005

Abstract

The Perl programming language has several features that predispose it to use by casual programmers[25]. These features also make it safer to use than low-level languages like C, in a security sense. However, Perl uses a very permissive dynamic typing system that prevents the compiler from identifying type errors at compile time[1]. In this paper, we discuss a type system for Perl that should help eliminate these errors before they become a problem. Perl has some features that preclude soundness and limit the usefulness of the current implementation, so future work to rectify these problems is discussed as well.

1 Introduction

1.1 Features of Perl

There are four features of Perl that make it useful from a security standpoint: references, arrays, hashes (or associative arrays), and Taint mode. These help to secure Perl by preventing misuse, discouraging creation of bad data structures, and preventing potentially dangerous strings from leaking in to sensitive functions.

References in Perl are a little bit like pointers in C, where references can be nested arbitrarily. However, there is no explicit allocation or de-allocation: dynamic memory is reference counted. References are made to global variables as well as lexically scoped variables in such a way that it is impossible to have a dangling reference to some part of memory that might get re-used later. For example, if a reference is made to a lexically scoped variable in a function, that instance is made permanent up to the reference count dropping to zero. Subsequent calls to that function will not interfere with the referenced memory. Moreover, it is impossible to create references to arbitrary locations in memory. The programmer can create references through two mechanisms: references to existing variables, and references to literals. This restriction on reference creation prevents dangerous misuse of memory.

Indices in to Perl arrays can be arbitrary. There is no such thing as an out-of-bounds index in to a Perl array, since Perl will resize the array as necessary at run-time if the program writes beyond the current end of the array. Reading an undefined index

[1]Perl is a scripting language in that it reads programs in plain text form and executes them. However, it compiles the programs in to an intermediate form before execution.

```
our $foo;
if (int(rand(2)) % 2) {
   $foo = 1;
} else {
   $foo = "foo";
}
print($foo + 1, "\n");
```

Figure 1: In half of the executions, `$foo` holds the value “foo” which is coerced to zero at the addition.

from an array will simply return the undefined value. Hashes in Perl behave the same way, except they are indexed by strings. Hashes serve as the stand-in for compound data types, like the `struct` in C. There is a slight performance penalty for the hash table lookup, but it means that hash instances can stand in for each other in a memory-safe (but type-unsafe) way. With references, arrays, and hashes, the programmer can construct complicated but memory-safe data structures with relative ease.

Taint mode is the most prominent security feature[6]. As part of Perl’s practical nature, programmers must be able to invoke external programs. There is an inherent danger to external invocation, since this mechanism often goes through the underlying shell’s interpretation. Thus, data consumed by the Perl program may have shell metacharacters in it that cause these invocation mechanisms to do the wrong thing. For example, suppose a Perl program reads the following value in to `$foo`:

```
foo;mail -s "" foo@foo.com < /etc/shadow
```

Now, suppose the Perl program executes the following statement:

```
system("ls $foo");
```

If the Perl program is running with root authority on a Unix system, this will result in unintended consequences. The shadow password file will be mailed to an off-site address. This is because the shell treats the semicolon character as a statement separator.

Perl’s solution to unchecked user data is to mark all incoming data, including that from environment variables, as tainted. Unchecked data cannot be used in the invocation. Instead, it must be matched with a regular expression. References to the parenthetical subexpressions of the match are then untainted. For instance, the variable `$1` is untainted after matching input with `/^([a-zA-Z0-9]*)/`. The tainted input in the example would fail to match that expression. Taint is checked at run time, and thus taint mode is an excellent example of execution monitoring in practice.

1.2 Dangers

Safe references, arrays, and hashes are requirements in Perl because typing in Perl is otherwise permissive. Strings, numbers, and references in Perl are all stored as scalar

```
our $foo;
if (int(rand(2)) % 2) {
   # ref to integer
   $foo = \1;
} else {
   # ref to array
   $foo = [];
}
# dereference and add
print($$foo + 1, "\n");
```

Figure 2: In half of the executions, `$foo` holds a reference to an anoymous array which can not be added at the addition.

variables, stated in code by prepending a `$` symbol to the name. This is similar to arrays and hashes, which have `@` and `%` prepended to their names, respectively. When a scalar value of some type is used in a different context, the scalar value is automatically coerced to the other type. For instance, strings are converted syntactically to numbers when used in a number context: if a prefix of the string represents a number, then that number is used. Otherwise, the string is converted to zero. The example in figure 1 illustrates this effect. Perl also allows the use of uninitialized and undeclared variables. There are also some outright fatal errors that the Perl compiler cannot catch, particularly with respect to references. Figure 2 is a valid Perl program that crashes under certain circumstances.

1.3 Current Solutions

At the current time, Perl has some features that be used on an advisory basis to help mitigate some of these weaknesses. The most prominent is the `-w` flag, which issues warnings when type coercions are done and uninitialized variables are used at run-time. It will also catch a few coercions at compile time, but only in the most simple cases. For instance, neither of the previous examples is caught by the `-w` flag at compile time. Perl also has the `use strict` pragma which helps catch the use of undeclared variables at compile time, as well as prevent the use of symbolic references[2]at run-time, among other things. These tools are required for writing non-trivial Perl programs, but they only go so far to ensure safety.

2 Type Inference

The remedy we propose is a type inference system for Perl. However, there is no formal grammar for Perl aside from the implementation itself. To this end, we have constructed a program that uses the compiler back end [9] to type check the Perl opcode

[2]In addition to conventional references, discussed previously, it is possible to reference variables by name instead of address, a dangerous and archaic holdover from previous versions of Perl

$$\begin{array}{rcl}
\tau & ::= & \mathrm{M}\,\mu \\
\mu & ::= & \mathrm{H}\,\eta \,|\, \mathrm{K}\,\kappa \,|\, \mathrm{AV} \,|\, \mathrm{HV} \,|\, \mathrm{CV} \,|\, \mathrm{IO} \\
\eta & ::= & : \mathrm{AV}, \mathrm{HV}, \mathrm{CV}, \mathrm{IO}, \kappa \\
\kappa & ::= & \mathrm{P}\,\tau \,|\, \mathrm{N}\,\nu \,|\, \mathrm{PV} \\
\nu & ::= & \mathrm{IV} \,|\, \mathrm{DV}
\end{array}$$

Figure 3: The type language for complete types. This is a regular language.

tree that is produced by the compiler before the main body of the program is executed. This is an important distinction: we type check the post-compiler output, not the Perl itself. However, the Perl virtual machine uses a large number of opcodes (about 350) that reflect the operators as used in the Perl language. This means that the post-compiler output is much closer to the source than a program in another language compiled to assembly.

2.1 Perl Type System

At run time, the Perl type system encompases many different sub-types [3]. At compile time, however, this complexity is hidden because constants have a restricted range of types, and little specific type information is stored in variables themselves. The Perl type system is complicated by the use of two separate types of symbol table (the global symbol table, and the per-function pad table). The global symbol table is stored internally with the Perl hash type, where variables are indexed by their name. Each value in this global symbol table is a typeglob, where one name refers to several different values of different types. For instance, the name `foo` in the global symbol table has a scalar (`$foo`), array (`foo`), hash (`%foo`), I/O file handle (`foo`), and code values (`&foo`) associated with it. For this reason, a typeglob acts like a reference, and can be used as such. Thus, Perl also has two types of references [5].

2.2 Static Type Inference

We propose an overlay type system that reflects the information about the program available at compile time. We can freely treat the `PVxx` (a pointer to something) types from the Perl run-time as regular references, since this is how the underlying opcodes treat them. Thus, this overlay type system does not directly exhibit the structure of the Perl type system. For instance, in order to disambiguate integers from double floating point values, we have IV and DV terminal types, whereas Perl has IV and NV types, where a NV can be an integer or a double.

2.2.1 Type Language

The type system for fully qualified types is illustrated in figure 3. In our overlay type language, any Perl type can be expressed as a fully qualified type in our lan-

Perl Type	Overlay Type	Meaning
AV	$\mathrm{M\,AV}$	Array
HV	$\mathrm{M\,HV}$	Hash (associative array)
CV	$\mathrm{M\,CV}$	Code (subroutine, usually)
IO	$\mathrm{M\,IO}$	File Handle
PV	$\mathrm{M\,K\,PV}$	String
IV	$\mathrm{M\,K\,N\,IV}$	Integer
NV	$\mathrm{M\,K\,N\,DV}$	Double Floating Point

Table 1: Terminal Types and their Meaning

$$\begin{array}{rcl}
\tau & ::= & \mathrm{M}\,\mu | \alpha_\tau \\
\mu & ::= & \mathrm{H}\,\eta | \,\mathrm{K}\,\kappa | \,\mathrm{AV}\,|\,\mathrm{HV}\,|\,\mathrm{CV}\,|\,\mathrm{IO} \\
\eta & ::= & :\mathrm{AV},\mathrm{HV},\mathrm{CV},\mathrm{IO},\kappa \\
\kappa & ::= & \mathrm{P}\,\tau | \,\mathrm{N}\,\nu | \,\mathrm{PV}\,|\alpha_\kappa \\
\nu & ::= & \mathrm{IV}\,|\,\mathrm{DV}\,|\alpha_\nu
\end{array}$$

Figure 4: The type language after introducing type variables.

guage. For instance, table 1 shows Perl types and how they are expressed in our type language. In our type language, typeglobs correspond to the η non-terminal ($\mathrm{M\,H} : \mathrm{AV}, \mathrm{HV}, \mathrm{CV}, \mathrm{IO}, \kappa$). References are expressed similarly. For instance, a reference to a PV is written thusly: $\mathrm{M\,K\,P\,M\,K\,PV}$.

Ideally, a type inference system for Perl would use a Soft Typing system with union types [11, 7]. However, these systems can be very complicated and inefficient. Instead, we have constructed our type lanaguage so that we can introduce type variables in very specific places (see figure 4). This way, we can represent unqualified numbers ($\mathrm{M\,K\,N}\,\alpha_\nu$), generic scalars ($\mathrm{M\,K}\,\alpha_\kappa$), and references to anything ($\mathrm{M\,K\,P}\,\alpha_\tau$) without the need for expensive union types. For instance, an operator that yields a generic scalar can be freely unified with another operator that requires an integer.

2.2.2 Type Unification

Our type unification algorithm works much like that presented in [12], without type polymorphism (since we only type check single functions). The key feature of type variables introduced at specific structural points means that our unification algorithm does need to work somewhat differently. In the base case, where at least one of τ_1 and τ_2 are type variables, unification works as expected. However, if they are both partially concrete types, we compare down the type string until they differ, or one is a type variable.

For example, as shown in figure 5, the unification for a string ($\mathrm{M\,K\,PV}$) and a generic scalar ($\mathrm{M\,K}\,\beta$) would proceed as follows:

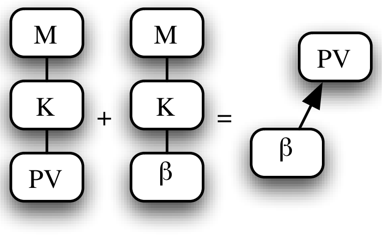

Figure 5: Unifying a string ($\mathrm{M\,K\,PV}$) with a generic scalar ($\mathrm{M\,K}\,\beta$) yields β unified with PV.

$$\frac{\Gamma \vdash t_1 : \mathrm{M\,AV} \quad \Gamma \vdash t_2 : \mathrm{M\,K\,N\,IV} \quad \alpha\,\mathrm{fresh}}{\Gamma \vdash \mathrm{AELEM}(t_1, t_2) : \mathrm{M\,K}\,\alpha_\kappa}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{M\,K\,N\,IV} \quad \Gamma \vdash t_2 : \mathrm{M\,K\,N\,IV}}{\Gamma \vdash \mathrm{I\_ADD}(t_1, t_2) : \mathrm{M\,K\,N\,IV}}$$

Figure 6: Type rules for several Perl operators.

1. Compare M from both types
2. Compare K from both types
3. Compare PV and β from both types
4. Unify PV and β.

2.2.3 Type Rules

Using this type language, most type rules are straightforward. For example, the rules `AELEM` (array access) and `I_ADD`[3]are unambiguous and obvious (see figure 6. This is what we desire in type rules.

However, there are some operators that defy simple, unambiguous rules. Binary operators that work on numbers generally do different things depending on what kind of operands they are given (see figure 7 for `ADD`). This may include coercions from integer values to floating point values. In a type inference system with incomplete types, these rules are not deterministic, since it is unclear in an $add(\mathrm{M\,K\,N}\,\beta, \mathrm{M\,K\,N}\,\gamma)$ situation what types β and γ should be inferred. This presents a problem, since we would like to infer types for numbers before doing things like using one as an index to an array. To this end, we have implemented the type rules so that we can at least infer a partial type. In the stated ambiguous case, β would be unified with γ, and the whole operation would be typed as $add(\mathrm{M\,K\,N}\,\beta, \mathrm{M\,K\,N}\,\gamma) : \mathrm{M\,K\,N}\,\beta$. Later, β will be unified with IV if the result of the operation is used as an index to an array.

[3]`I_ADD` is explicit integer addition. This opcode is only generated with the `use integer` pragma.

$$\frac{\Gamma \vdash t_1 : \mathrm{MKNDV} \quad \Gamma \vdash t_2 : \mathrm{MKNDV}}{\Gamma \vdash \mathrm{ADD}(t_1, t_2) : \mathrm{MKNDV}}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{MKNIV} \quad \Gamma \vdash t_2 : \mathrm{MKNDV}}{\Gamma \vdash \mathrm{ADD}(t_1, t_2) : \mathrm{MKNDV}}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{MKNDV} \quad \Gamma \vdash t_2 : \mathrm{MKNIV}}{\Gamma \vdash \mathrm{ADD}(t_1, t_2) : \mathrm{MKNDV}}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{MKNIV} \quad \Gamma \vdash t_2 : \mathrm{MKNIV}}{\Gamma \vdash \mathrm{ADD}(t_1, t_2) : \mathrm{MKNIV}}$$

Figure 7: Type rules for `ADD`.

2.2.4 Difficulties

As previously discussed, there are some ambiguities that interfere with inference. In addition to the binary numeric operators, reference operators treat typeglobs and references to typeglobs the same. That is, when they run, they will silently dereference the reference to a typeglob before executing as expected. There are some operators (such as `OPEN`) that are so overloaded that no type information can be inferred about their operands.

Beyond these challenges, inference in full Perl cannot be sound. Perl idioms like “`eval "string"`” and “`do "filename"`” preclude type safety since they cause the Perl interpreter to compile and run code while the program is running. The example in figure 8 breaks type safety in a way that cannot be detected at compile time. Furthermore, in the current implementation, type information is lost when storing and retrieving through aggregate types. For instance, type preservation fails when a reference is written to the first element of an array, and then a read from that position is later inferred as an integer. However, even if preservation could be shown, there are no formal operational semantics to prove progress. The Perl language semantics are defined by the C implementation.

These problems highlight two needs: first, a useful type system probably requires a better way to handle the aggregate data types. Second, it is desireble to have a Featherweight Perl, modelled after Featherweight Java [20]. This is an ironic task, since the designers have explicitly stated that “[Perl] is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)” [24].

```
our $foo;
if (int(rand(2)) % 2) {
   # ref to integer
   $foo = \1;
} else {
   # ref to array
   eval "$foo = [];"
}
print($$foo + 1, "\n");
```

Figure 8: This code fails in half of its executions, and there is no way for the problem to be detected at compile-time in the general case (since the argument to eval may be a variable).

Change	`stow`	`cookie.lib`	Total
Empty Array Test	10	2	12
Broken Type Names[4]	7	0	7
Undefined versus PV	4	0	4
IV to PV Coercion	0	3	3
Empty String Test	2	0	2
Total	23	5	28

Table 2: Summary of Changes

3 Experimental Results

We ran the type inference system against three Perl programs. We hoped to find some bugs, but our analysis required merely a few inconsequential changes to run. The programs otherwise came back clean. This is somewhat disappointing, but it may be the case that none of the programs examined had type related errors, or that all of the type related errors are obscured through aggregate data structures or are only revealed through interprocedural analysis. It also might suggest that the need for type checking in Perl is overstated.

We examined `stow` [16], `cookie.lib` [26], `gradeorama.cgi` [22], and the `Typcheck` program itself. The `stow` program is a command line utility for managing software versions through symbolic links. This bears particular scrutiny, since usually someone with root authority uses the product of off-site users as input (their installed programs), and affects change in a system-wide sensitive place (such as `/usr/local`, on a UNIX host). The other programs, `cookie.lib` and `gradeorama.cgi`, are a library for manipulating cookies in a CGI context and a gradebook application for secondary school teachers, respectively.

We ran the typechecker to check the main body of the program as well as all subroutines. The typechecker does not do general interprocedural analysis, but it does

[4]This reflects a bug in the type checker, but the fix was a much more readable English name.

Program	Total Time	User	System	Opcodes	Lines
`cookie.lib`	1.09692	0.9704	0.0864	904	546
`stow`	2.87484	2.6278	0.125	3145	547
`gradeorama.cgi`	8.55935	8.2027	0.2	7845	1862
`Typecheck`	21.11438	20.0186	0.4288	9479	1868

Table 3: Summary of Performance Results. Times are expressed in seconds.

ensure that global variables are used consistently in all code checked.

We did not find any substantive errors in thes programs. We made some changes to `stow` and `cookie.lib` so that they would type check without error. We did make any changes to `gradeorama.cgi`, which typechecked without error. Most of the changes (12 out of 28) were changing the guards on loops and branches that tested for an empty array by using the array itself. We changed it to a test on the length of the array instead. The rest of the changes were equally unimportant with respect to the correctness of the programs. A summary of the changes is in table 2. We did find one error in the `Typecheck` program that could not be fixed because we are using a language misfeature for reflection.

All experiments were run on an Apple Macintosh PowerBook with an 864MHz PowerPC G4 processor and 640MB of RAM, running MacOS X version 10.3.9. The version of Perl used was 5.8.1, which is the default version packaged with that release of the operating system. All programs were typechecked 100 times in a row, and the averages are summarized in table 3. The run time is significantly more than the overhead to compile the program, which is less than two tenths of a second in all cases. The `Typecheck` program is a disturbing outlier, but this is probably due to the deeply nested nature of the `typeOp` function contained therein.

4 Future Work

Clearly, this tool needs more work to be truly useful to Perl programmers. There are several Perl features that need support, including aggregate data structures and Perl objects. There is also a demand for functionality beyond that available through inference on unannotated code.

4.1 Aggregate Data Structures

As stated previously, reads and writes through aggregate data structures cause values to lose their types, as far as the type inference algorithm is concerned. It would be desireable to retain as much type information as possible.

4.1.1 Arrays

Arrays are critical for typing Perl; many things are treated as arrays, such as function arguments. Perl goes much farther than the type covariant arrays of Java and C#. Perl arrays can hold any variety of scalar value in any order. For instance, the following is a legal literal assignment for a Perl array:

```
@foo = (1, "2", \3, IO::File->new("</etc/services"))
```

This array might be typed as:

$$(\mathrm{M\,K\,N\,IV}, \mathrm{M\,K\,PV}, \mathrm{M\,K\,P\,M\,K\,N\,IV}, \mathrm{M\,K\,P\,M\,HV}(\mathrm{IO::File}))$$

However, precise type tuples like this may not be possible or even desirable, in general. Instead, we propose a regular type for arrays. The alphabet for these regular expressions would be countably infinite (due to references), but the type atoms themselves would be regular. For example, the following might be used as the type of arguments that the Perl `print` operator would take. A candidate list of arguments would be matched against this using regular expression matching, if the candidate list is not a regular type. Alternatively, if the candidate is a regular type, it could be checked with a regular expression subset examination. The following type would preclude printing references coerced to strings:

$$((\mathrm{M\,K\,P\,M\,HV}(\mathrm{IO::File})|\epsilon) \cdot (\mathrm{M\,K\,N}\,\alpha_\nu | \,\mathrm{M\,K\,PV})*)$$

A common Perl idiom is to treat arrays like a deque with the `push`, `pop`, `shift`, and `unshift` operators. This, along with array concatenation, could support this effort. There is already a body of work for using regular expression types with XML [18]. At this time, though, it is unclear if regular expression types would be useful for Perl, since they may still devolve in to something like a Kleene star around a union of many types, thus rendering the expression effectively useless.

4.1.2 Hashes

Hashes in Perl work much the same way as arrays. However, there are certain idioms that hashes are used for that would make incomplete typing still very useful. As stated in the introduction, hashes are used for compound data types with named members, much like the `struct` in C. If all we do is track types for members with constant names, we will be able to do useful type inference. It might be possible to describe a subtyping relationship in terms of the requirements that a function has for the members of a hash that must be available. That is, in Perl, any hash is effectively equivalent to any other hash until the program starts reading and writing entries. To realize this in the type system, we can require that a hash value have the value at the given index at the point of use, but otherwise make no restrictions on the hash. For instance, a function that only uses the name field of a hash may freely use hashes that describe employees and customers separately without any explicit relationship between the two different effective hash types.

This is particularly important for Perl objects because objects are just references to normal Perl data types with “magic” [4]. In most cases, a Perl object is a reference to a hash value, thus making at least incomplete typing of hashes important for typing Perl objects.

4.2 Object-Oriented Perl

Perl Objects are a necessity for a type inference tool to be useful. Most recently published Perl programs are written in an object oriented style, and most modules in CPAN are written as objects [2]. In fact, mining CPAN was once considered for mining material for the experiments discussed in this paper, but their inherent object oriented nature makes attempting to type them less useful.

As mentioned in the previous section, typing hashes would be important for typing objects in Perl. An implicit subtyping relationship could be useful here for method calls through objects, as well. However, such a thing is not required because Perl does have an explicit inheritance mechanism. In fact, it is probably desirable to use only the inheritance mechanism when evaluating subtyping relationships because legal use of objects without a relationship may still be an error.

4.2.1 Recursive Types

As in any similar language, objects can be used to construct complex recursive structures. For instance, the implementation written for this paper uses recusive data structures to represent references. If the type analysis were extended to interprocedural analysis, the type check would probably fail due to the occurs check. Thus, there is a need to support recursive types [8, 17].

4.3 Flow Sensitivity

Support for regular array types requires a flow sensitive analysis, because the construction of the regular expression requires some knowledge of the order of operations. Moreover, flow sensitive analysis would make more precise types, and make the type checker less sensitive to the re-use of variables (though it could be argued that programmers should not be re-using variables, particularly in a type-inconsistent way). A data flow harness has already been developed for Perl in C [21]. Re-writing the harness in Perl for use with the type checking module is an obvious next step for supporting flow-sensitve type analysis. However, the analysis might be terribly inefficient because global type information would have to be retained on a basic block basis, instead of a single instance. Some sort of copy-on-write scheme for the type analysis is a requirement.

4.4 Type Qualifiers

Type Qualifiers [14] are a desirable feature for type inference systems, because they allow tracking relevant information along with types that may not be available directly from the type itself. In a limited sense, Perl implements a type qualifier system with a subtyping relationship that prevents tainted data from flowing in to sensitive functions. However, taint mode only works at run time, and its implementation is extremely intrusive in to Perl. The Perl source is littered with very specific Taint mode machinery, so adapting the taint mechanism to other type qualifier systems is impractical. Thus, adding type qualifiers to the current type inference system would be useful. For instance, the inheritance relationships among objects could be used to establish the subtyping relationship, and then object type could be tracked as a qualifier on a reference type to implement type checking for object oriented Perl. Beyond that, an attributes system [1] has been introduced with recent versions of Perl. These annotations can be used to mark functions and local variables. Through this mechanism it may be possible to annotate argument and return values for Perl functions.

5 Related Work

5.1 Perl

The only tool available for checking Perl programs, beyond the default flags and pragmas, is the `B::Lint` [10] module that is part of the compiler back-end suite distributed with Perl. This does a number of analyses, including warning when an array is used in a scalar context. As stated previously, there has been a data flow analysis tool developed for Perl [21]. However, that tool currently only implements a very specific bug-finding algorithm. Lastly, there has been a lot of chatter on the Perl 6 mailing list about Type Inference for Perl 6, and a Request For Comments has been written about it [13]. However, this work appears to still be in its infant stages, and does little for current Perl 5 programs.

5.2 Analysis for Other Languages

There are numerous tools for static analysis on other languages. The two that the author is most familiar with are CQUAL [23, 15] and FindBugs [19]. CQUAL, in particular, implements a type qualifier system for C. This is a good model for implementing type qualifiers for Perl.

6 Conclusion

Perl has many nice security features. However, these features are necessary due to Perl's dynamic type system. Perl allows type-inconsistent use of values and will not alert the user until the program runs. Type inconsistency can be mitigated through the application of a type inference system; we implemented one that works. Unfortunately, aspects of Perl limit precise type inference. Furthermore, limitations of the implementation constrain its overall usefulness. Thus, there is plenty of room for further development of a static type inference system for Perl.

References

[1] *attributes - get/set subroutine or variable attributes.* `attributes(3pm)`, part of the Perl Programmers Reference Guide.

[2] Comprehensive Perl Archive Network (CPAN). `http://www.cpan.org`.

[3] *perlguts - Introduction to the Perl API*. `perguts(1)`, part of the Perl Programmers Reference Guide.

[4] *perlobj - Perl objects*. `perlobj(1)`, part of the Perl Programmers Reference Guide.

[5] *perlref - Perl references and nested data structures*. `perref(1)`, part of the Perl Programmers Reference Guide.

[6] *perlsec - Perl security*. `perlsec(1)`, part of the Perl Programmers Reference Guide.

[7] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–118, New York, NY, USA, 1991. ACM Press.

[9] Malcolm Beattie. *B - The Perl Compiler*. `B(3pm)`, part of the Perl Programmers Reference Guide.

[10] Malcolm Beattie. *B::Lint - Perl lint*. `B::Lint(3pm)`, part of the Perl Programmers Reference Guide.

[11] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[12] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[13] Steve Fink. type inference, August 2000. `http://dev.perl.org/perl6/rfc/4.html`.

[14] Jeffrey S. Foster, Manuel F&#228;hndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[15] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[16] Bob Glickstein. GNU stow. `http://www.gnu.org/software/stow/stow.html`.

[17] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[18] Haruo Hosoya, J&#233;r&#244;me Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[19] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136. ACM Press, 2004.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[21] Gary L. Jackson II. Static analysis of perl. Project for CMSC631, December 2004.

[22] Kristina L. Pfaff-Harris. GradeORama: Perl/CGI Online Gradebook, 2004. `http://tesol.net/scripts/GradeORama/`.

[23] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, August 2001.

[24] Larry Wall and et. al. *perl - Practical Extraction and Report Language*.

[25] Larry Wall and Mike Loukides. *Programming Perl*. O'Reilly & Associates, Inc., 2000.

[26] Matthew M. Wright. Matt's Script Archive: HTTP Cookie Library, 1996. `http://www.scriptarchive.com/cookielib.html`.

A Implementation Notes

The implementation strongly reflects the type language as expressed in figure 4. All capital Greek letters (M, H, K, P, and N), all terminal types (AV, HV, CV, IO, IV, DV, and PV), and the variable type (α) are implemented as Perl classes. A constructed type is a linked list of these objects, and the constructors prevent illegal types from being joined together. The type inference algorithm itself is a giant "`if ...elsif ...else`" implementation of the Damas-Milner algorithm [12] that unifies candidate types with immutable constant types and fresh incomplete types, as necessary.