Third Workshop on Script to Program Evolution STOP 2012

This year marks the third edition of the STOP workshop series, and a colocation with ECOOP and PLDI in Beijing, China. For this year's STOP, the goal was to take stock of the impressive progress on many fronts in our young area and to hear reports from our colleagues on their current progress and future directions. Therefore, we solicited short submissions, collected here, in preparation for lively discussion at the workshop.

The continued success of STOP is a testament to the hard work of many people. I would like to especially thank Adam Welc, Jan Vitek, and Arjun Guha for their assistance in the organization, as well as all the members of the program committee.

Sam Tobin-Hochstadt (chair)

Status Report: Dependent Types for JavaScript

Ravi Chugh
University of California, San Diego
rchugh@cs.ucsd.edu

David Herman
Mozilla Research
dherman@mozilla.com

Ranjit Jhala
University of California, San Diego
jhala@cs.ucsd.edu

We are developing Dependent JavaScript (DJS), an explicitly-typed dialect of a large JavaScript subset that features mutable, prototype-based objects and arrays as well as precise control-flow reasoning. We *desugar* DJS programs to a core lambda-calculus with explicit references following in the style of λ_{JS} [3]. Our new type system operates on desugared programs, building upon techniques from System D [2] and Alias Types [4]. With our preliminary implementation, we demonstrate that DJS is expressive enough to reason about a variety of tricky idioms found in small examples drawn from several sources, including the popular book *JavaScript: The Good Parts* and the SunSpider benchmark suite. In this report, we provide a brief overview of DJS; more details can be found in [1].

Path Sensitivity. Consider the following function where the type annotation says that if the input is a number, then so is the return value, and otherwise it's a boolean; the "if-then-else" macro **ite** $p\ q_1\ q_2$ abbreviates the formula $(p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2)$.

```
//: x:Top → {ν | ite Num(x) Num(ν) Bool(ν)}
function negate(x) {
  return (typeof x == "number") ? 0 - x : !x;
}
```

When checking the true case of the conditional, DJS tracks that `x` is a number. Because the subtraction also produces a number, it concludes that the return value has type $\{\nu \mid Num(x) \Rightarrow Num(\nu)\}$. In the false case, `x` is an arbitrary, non-number value, which is safe to use because the JavaScript negate operator inverts the "truthiness" of *any* value, not just booleans. So, the return value has type $\{\nu \mid \neg Num(x) \Rightarrow Bool(\nu)\}$. By combining the types of values stored in `x` along both branches, DJS verifies that the return type satisfies its specification.

Refinement Types. Even the simple example above requires sophisticated propositional and equational reasoning that depends on program *values*. In DJS, we employ *refinement types* to encode these relationships and use an SMT solver to discharge logical validity queries that arise during (sub)type checking. Refinement types are quite expressive but, by using formulas drawn from a *decidable* logic, once the programmer has provided annotations on functions, type checking proceeds automatically. In comparison, more expressive *dependent* type systems like Coq rely on the programmer or heuristics to interactively discharge proof obligations.

Primitive Operators. In DJS, we use refinements to assign precise types to primitive operators; we show a few below.

$$\begin{aligned}
\texttt{typeof} &:: x{:}\,Top \to \{\nu \mid \nu = tag(x)\} \\
\texttt{!} &:: x{:}\,Top \to \{\nu \mid \textbf{ite}\ falsy(x)\ (\nu = \texttt{true})\ (\nu = \texttt{false})\} \\
\texttt{\&\&} &:: x{:}\,Top \to y{:}\,Top \to \{\nu \mid \textbf{ite}\ falsy(x)\ (\nu = x)\ (\nu = y)\} \\
\texttt{||} &:: x{:}\,Top \to y{:}\,Top \to \{\nu \mid \textbf{ite}\ falsy(x)\ (\nu = y)\ (\nu = x)\}
\end{aligned}$$

The above types allow DJS to reason about `negate` and idioms like `if (x && x.f)` to guard key lookups and `x = x || default` to set default values. Refinement types provide the flexibility to choose more restrictive types for operators, if desired, to statically prevent implicit coercions, which often lead to subtle programming errors. For example, we can restrict the negation operator to boolean values as follows.

$$\texttt{!} :: x{:}\,Bool \to \{\nu \mid \textbf{ite}\ (x = \texttt{false})\ (\nu = \texttt{true})\ (\nu = \texttt{false})\}$$

Flow Sensitivity. Consider the following function that is like `negate` but first assigns the eventual result in the variable `x`.

```
//: x:Top → {ν | ite Num(x) Num(ν) Bool(ν)}
function also_negate(x) {
  x = (typeof x == "number") ? 0 - x : !x;
  return x;
}
```

To precisely reason about the different types of values stored in the (imperative) variable `x`, DJS maintains a flow-sensitive heap that can be *strongly updated* at each program point. As a result, DJS tracks that the updated value of `x` along the true case is $\{\nu \mid Num(x) \Rightarrow Num(\nu)\}$ (where x is the formal parameter initially stored in `x`) and along the false case is $\{\nu \mid \neg Num(x) \Rightarrow Bool(\nu)\}$. Thus, as before, DJS verifies that the return value (the new value of `x`) satisfies the specification.

Objects. JavaScript objects make heavy use of property extension and prototype inheritance to transitively resolve lookups.

```
var parent = {last: " Smith"};
var child = Object.create(parent);
child.first = "Bob";
child.first + child.last; // "Bob Smith"
```

For object extension, strong updates allow DJS to track that the "`first`" property is added to `child`. For prototypes, DJS precisely tracks parent links between objects in the heap, and *unrolls* prototype chains to match the semantics of object operations. For example, the type of the value retrieved by `child.last` is

$$\begin{aligned}
\{\ \nu \mid\ &\textbf{if}\ has(\texttt{child}, \text{“}\texttt{last}\text{”})\ \textbf{then}\ \nu = sel(\texttt{child}, \text{“}\texttt{last}\text{”}) \\
&\textbf{elif}\ has(\texttt{parent}, \text{“}\texttt{last}\text{”})\ \textbf{then}\ \nu = sel(\texttt{parent}, \text{“}\texttt{last}\text{”}) \\
&\textbf{else}\ \nu = \texttt{undefined}\ \}
\end{aligned}$$

which is a subtype of $\{\nu \mid \nu = sel(\texttt{parent}, \text{“}\texttt{last}\text{”})\}$ given what we know about `child` and `parent`. Furthermore, we use *uninterpreted heap symbols* to reason about portions of the heap that are statically unknown. This allows DJS to verify that the property lookup in `if (k in x) x[k]` does *not* return `undefined` (unless the type of `x[k]` includes `undefined`, of course) even when *nothing* is known about the prototype chain of `x`.

Arrays as Arrays. Arrays are (mostly) ordinary prototype-based objects with string keys, but JavaScript programmers (and optimizing JIT compilers) commonly treat arrays as if they are traditional "packed" arrays with integer "indices" zero to "size" minus one. DJS reconciles this discrepancy by maintaining the following invariants about every array `a` $:: Arr(T)$.

1. `a` contains the special "`length`" key.
2. All other "own" keys of `a` are (strings that coerce to) integers.
3. For all integers i, either `a` maps the key i to a value of type T, or it has no binding for i.
4. All inherited keys of `a` are "safe" (*i.e.* non-integer) strings.

Furthermore, we use the uninterpreted predicate $packed($`a`$)$ to describe arrays that also satisfy the following property, where $len($`a`$)$ is an uninterpreted function symbol.

5. For all integers i, if i is between zero and $len($`a`$)$ minus one, then `a` maps i to a value of type T. Otherwise, `a` has no binding for i.

These invariants allow DJS to reason locally (without considering the prototype chain of `a`) that for any integer, `a[i]` produces a value of type $\{\nu \,|\, \nu :: T \vee \nu = \texttt{undefined}\}$, and that if $0 \leq$ `i` $< len($`a`$)$, then `a[i]` *definitely* has type T. We assign types to array-manipulating operations, including the `Array.prototype.push` and `Array.prototype.pop` functions that all arrays inherit, to maintain these invariants and treat packed arrays precisely when possible.

Tuples. Arrays are used as finite tuples in several idiomatic ways.

```
var a0 = [0, 1, 2];
var a1 = []; a1[0] = 0; a1[1] = 1; a1[2] = 2;
var a2 = []; a2.push(0); a2.push(1); a2.push(2);
```

For `a1` and `a2`, DJS is able to track that the array updates — even when going through the `Array.prototype.push` native function that is inherited by `a` — maintain the invariant that the arrays are packed. Thus, each of the arrays has the following type.

$$\{\nu \,|\, \nu :: Arr(Int) \wedge packed(\nu) \wedge len(\nu) = 3\}$$

Benchmarks. We are actively working on our implementation (available at `ravichugh.com/nested`). So far, we have tested on 300 lines of unannotated benchmarks from several sources including *JavaScript: The Good Parts* and the SunSpider and V8 microbenchmark suites. Figure 1 summarizes our current results, where for each example: "Un" is the number of (non-whitespace, non-comment) lines of code in the *unannotated* benchmark; "Ann" is the lines of code in the annotated DJS version (including comments because they contain DJS annotations); "Time" is the running time rounded to the nearest second; and "Queries" is the number of validity queries issued to Z3 during type checking.

Taken together, the set of benchmarks rely on the gamut of features in the type system, requiring type invariants that describe relationships between parent and child objects, between the contents of imperative variables and arrays across iterations of a loop, and intersections of function types to encode control-flow invariants.

Annotation Burden and Running Time. As Figure 1 shows, our annotated benchmarks are approximately 1.7 times as large (70% overhead) as their unannotated versions on average. In our experience, a significant portion of the annotation burden is boilerplate — unrelated to the interesting typing invariants — that fall into a small number of patterns, which we have started to optimize.

Adapted Benchmark	**Un**	**Ann**	**Queries**	**Time**
JS: The Good Parts				
`prototypal`	18	36	731	2
`pseudoclassical`	15	23	706	2
`functional`	19	43	862	8
`parts`	11	20	605	3
SunSpider				
`string-fasta`	10	18	263	1
`access-binary-trees`	34	50	2389	23
`access-nbody`	129	201	4225	39
V8				
`splay`	17	36	571	1
Google Closure Library				
`typeOf`	15	31	1975	52
Other				
`negate`	9	9	296	1
`passengers`	9	19	310	3
`counter`	16	24	272	1
`dispatch`	4	8	219	1
Totals	306	518	13424	137

Figure 1. Benchmarks (Un: LOC without annotations; Ann: LOC with annotations; Queries: Number of Z3 queries; Time: Running time in seconds)

The running time of our type checker is acceptable for small examples, but less so as the number of queries to the SMT solver increases. We have not yet spent much effort to improve performance, but we have implemented a few optimizations that have already reduced the number of SMT queries. There is plenty of room for future work to further improve both the annotation overhead as well as performance.

Conclusion. We have found that the full range of features in DJS are indeed required, but that many examples fall into patterns that do not simultaneous exercise all features. Therefore, we believe that future work on desugaring and on type checking can treat common cases specially in order to reduce the annotation burden and running time, and fall back to the full expressiveness of the system when necessary. In addition, we are working to extend DJS with support for additional features, including more general support for recursive types, for the `apply` and `call` forms (often used, for example, to set up inheritance patterns), and variable-arity functions. We believe that Dependent JavaScript is a promising approach for supporting real-world dynamic languages like JavaScript.

References

[1] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. April 2012. `http://arxiv.org/abs/1112.4106`.

[2] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *POPL*, 2012.

[3] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

[4] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *ESOP*, 2000.

Minigrace: A progress report

Michael Homer
Victoria University of Wellington
mwh@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Abstract

Grace is a new language whose types are structural, gradual, and optional. Minigrace is a compiler for the Grace language. Minigrace itself was written with gradual types. We describe the implementation of the compiler and future challenges for it.

1. Introduction

Minigrace is a compiler for the Grace language [3]. Grace is a new object-oriented programming language aimed at education. The language supports both statically- and dynamically-typed code, along with a mixture of the two, and uses structural types as the basis for its static type system. Gradual types are intended to support different ways of teaching programming: an instructor may begin either with types or without, and introduce the other model later in the same language, with a smooth transition [1].

Minigrace implements the dynamically-typed part of the language and a static typechecker, and is able to run scripts directly or compile them to native code or JavaScript.

2. Background

The Grace language includes modern language features that have been useful in software practice, but have not been found together in the languages used for teaching. The language aims to include those features that are useful to learn without requiring that they be taught immediately or in any particular order. Instructors should have flexibility in structuring courses, and unintroduced features should not need to appear in code to be explained: the language tries to eliminate "boilerplate" code found in some languages.

Grace is a class-based language with single inheritance. Objects can be created from a class or from object literals, and as well as methods may contain mutable (**var**) and immutable (**def**) fields. Fields automatically define accessor methods of the same name.

Grace uses structural types, as in OCaml, in which the type of an object depends only on its methods, and not on explicit type names. Unlike most statically-typed class-based languages, a class in Grace does not automatically create a type of the same name. When a type is required the programmer must declare one explicitly containing the methods they want.

Types are optional; when omitted, code is dynamically-typed. When dynamic and static types mix gradual type checks at runtime implicitly preserve safety.

Classes and types can have unerased generic type parameters. A generic type with the same shape as a non-generic structural type will conform to that type, and vice-versa.

A Grace method name can have multiple words, each with an argument list, similar to Smalltalk. These "mixfix" methods allow the programmer to make the parameter roles clear.

Grace also supports first-class "blocks", or lambdas, chunks of code with deferred execution, written inside braces. Blocks return the value of the last expression in their body, while **return** statements return from the enclosing method. Methods and blocks create closures when they refer to names defined in the surrounding code.

```
method ifPositive(x : Number) then (b : Block) {
  if (x > 0) then {
    b.apply
  }
}
ifPositive 5 then {
  print "5 is positive"
}
```

Figure 1. User-defined control structure, using a mixfix method and a block

The built-in control structures of the language, **if**()**then**()**else**, **for**()**do**, and **while**()**do** are all defined as methods with multiple words taking blocks as arguments. A user-defined control structure could be provided in a library to replace or work alongside them without appearing any different, as in Figure 1. An instructor can replace or augment the structures presented to students.

3. Minigrace

Minigrace is a self-hosted compiler for Grace. The compiler targets native code (via C and LLVM bitcode) and JavaScript that runs in a commodity web browser. The compiler supports almost all of the dynamically-typed part of the language and a static typechecker.

The compiler self-hosts on all target platforms and can compile code with any target from any platform. The compiler makes heavy use of blocks and multi-word method names and sums to around 10,000 lines of code. As a self-hosted compiler targeting JavaScript, the compiler itself can be compiled to run in the web browser [4].

Minigrace contains a full static structural typechecker. Code to be compiled may be written fully statically-typed, fully dynamically-typed, or a mixture, and these are able to interact fully with each other. Dynamic code may have types added, and as these are structural types they are not overly intrusive, instead representing what the existing code already does.

Development of the compiler has followed a gradual typing approach. The compiler was originally prototyped entirely dynamically-typed, without a typechecker present. Later code has begun to include types.

The native compiler includes mark-and-sweep garbage collection and optional support for tail recursion. Programs may be written initially without concern for efficiency and function correctly.

3.1 Development

We first implemented a compiler/interpreter for a minimal subset of the language using the Parrot Compiler Toolkit, targeting the Parrot Virtual Machine. This compiler used the in-built grammar engine for parsing and Perl for linking code, and generated low-level

Parrot AST. We chose to begin here as we were considering Parrot as a potential target platform and wanted both to assess it for that purpose and to use its inbuilt parsing and compiling functionalities to bootstrap the compiler. Having brought this compiler to a minimal usable level we were able to begin writing Grace code for our later compiler.

The compiler has a fairly traditional structure with four phases: lexing, parsing, typechecking, and code generation. We wrote the lexer first, as our first real Grace program running on the bootstrap compiler, working on it until it could lex itself, and then the same for the parser. Once we were parsing correctly we implemented LLVM bitcode generation.

We chose to target LLVM at first as we were hoping to use some features (unwind instruction) to implement parts of Grace's semantics, but found these features to be less complete than hoped. We decided to implement a C backend as well, as increasingly much support code was in the accompanying C runtime library. We found C easier to debug, and switched to using it as our main backend.

The JavaScript backend was an experiment while waiting for other material to become available. We were able to produce a code generator sufficient to compile the compiler itself into runnable code. With this backend user code can run both natively and on the web. The compiler can be run in the browser for rapid prototyping without a full development infrastructure available.

Typechecking and identifier resolution occur together between parsing and code generation. The checker performs standard structural subtyping checks, but allows the Dynamic type to pass into other types freely. Only limited gradual run-time type checking occurs at method boundaries, although sufficient to catch many common errors.

A rule of the language is that types are optional, and have no effect on semantics, so the checker does not make any visible changes to the semantics of code. The programmer can disable typechecking altogether at compile time if they wish without affecting the behaviour of the program.

We developed the typechecker after we had a fully-functioning compiler using dynamically-typed code. We wanted to prototype and bootstrap the compiler quickly and did not want either to write the subtype computation and tree-walking checker or to deal with type errors that were not currently manifesting during this stage of development. After the integration of the typechecker new code could use static types, but still interact smoothly with the existing codebase.

After the typechecker was added we began adding partial type annotations to existing code when it was revised. These annotations caught some errors, although most of our bugs still were logical errors, rather than issues the typechecker could catch. These "drive-by" additions of types were also less effective than a more concerted strategy might have been: in many cases it worked out that an annotated section was only called from dynamically-typed code and only called out to other dynamic code, and so the benefit was limited before run-time checks were added. Gradual types were most useful after a bug had been found: we could add a type annotation to prevent the bug being reintroduced or to find other occurrences.

Another author extended the compiler to include a Java backend, generating Java source code. This was a separate module, though integrated into the codebase, and was fully statically-typed. The static module could find errors in itself while collaborating fully with dynamically-typed code in other modules.

4. Future challenges

4.1 Gradual structural typing

The combination of gradual and structural typing presents a problem for implementation. In some cases, incorrect types must be allowed to pass because they cannot be determined, even dynamically, to be invalid.

```
var o := object {
  method name {
    5
  }
}
type Named = {
  name −> String
}
method greet(n : Named) {
  print "Hello, {n.name}"
}
greet(o)
```

Figure 2. Example illustrating a difficulty with gradual structural typing

Given the Grace code in Figure 2, the call to greet must be allowed statically, as o is of type Dynamic. Gradual typing would insert a runtime typecheck at the call ensuring that the argument was of the Named type. However, that check must also pass – the name method returns Dynamic, which must be treated as conforming to String as well.

Only when the name method is called does the type mismatch appear, in fully static code. Detecting the problem requires checking at each assignment or return even in static code, and even that may not be adequate if the method is not called or the object passed elsewhere.

Alternatively, some sort of type-checking proxy object, a "chaperone" [5] implicitly inserted at the boundary, could limit the type checks to only those objects actually originating from dynamically-typed code. This approach would not address cases where the method is not called and raises additional problems of object identity and behaviour, potentially affecting semantics. The semantics required by the language are not yet fully specified, other than noting that static types should not affect runtime behaviour. Higher-order contracts [2] or other approaches are also possible. The correct approach is unclear, as the desired semantics are uncertain and error messages are especially important in an educational language.

Minigrace currently allows these cases to pass. Only when it is able to determine that a type error has occurred does it raise an error. When the type is uncertain it allows the program to execute, on the basis that otherwise valid programs would be rejected, and the dynamic type would have very restricted use in interacting with static code.

5. Conclusion

Minigrace is a compiler for the Grace language, written with a gradual typing approach. Code compiles to a variety of platforms and may incorporate a mix of static structural types and dynamic types.

References

[1] A. P. Black, K. B. Bruce, and J. Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, pages 201–204. ACM, 2010.

[2] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[3] The Grace programming language. http://gracelang.org/.

[4] M. Homer. Minigrace JavaScript backend. http://ecs.vuw.ac.nz/~mwh/minigrace/js/, 2011-2012.

[5] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for contracts on higher-order, stateful values. Technical Report NU-CCIS-12-01, Northeastern University, 2012.

Improving Tools for JavaScript Programmers

(Position Paper)

Esben Andreasen
Aarhus University
esbena@cs.au.dk

Asger Feldthaus
Aarhus University
asf@cs.au.dk

Simon Holm Jensen
Aarhus University
simonhj@cs.au.dk

Casper S. Jensen
Aarhus University
semadk@cs.au.dk

Peter A. Jonsson
Aarhus University
pjonsson@cs.au.dk

Magnus Madsen
Aarhus University
magnusm@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

ABSTRACT

We present an overview of three research projects that all aim to provide better tools for JavaScript web application programmers[1]: *TAJS*, which infers static type information for JavaScript applications using dataflow analysis; *JSRefactor*, which enables sound code refactorings; and *Artemis*, which provides high-coverage automated testing.

1. JAVASCRIPT PROGRAMMERS NEED BETTER TOOLS

JavaScript contains many dynamic features that allegedly ease the task of programming modern web applications. Most importantly, it has a flexible notion of objects: properties are added dynamically, the names of the properties are dynamically computed strings, the types of the properties are not fixed, and prototype relations between objects change during execution. An experimental study has shown that most dynamic features in JavaScript are widely used [10].

Such flexibility has a price. It becomes challenging to reason about the behavior of JavaScript programs without actually running them. To make matters worse, the language provides no encapsulation mechanisms, except for local variables in closures. For many kinds of programming errors that cause compilation errors or runtime errors in other languages, JavaScript programs keep on running, often with surprising consequences.

As a consequence, JavaScript programmers must rely on tedious testing to a much greater extent than necessary with statically typed languages. Additionally, it is difficult to foresee the consequences of modifications to the code, so code refactoring is rarely applied. Unlike the first scripts that appeared when JavaScript was introduced, today's JavaScript programs often contain thousands of lines of code, so it becomes increasingly important to develop better tool support for the JavaScript programmers.

[1]For more information about the projects and tools, see the website for Center for Advanced Software Analysis at `http://cs.au.dk/CASA`.

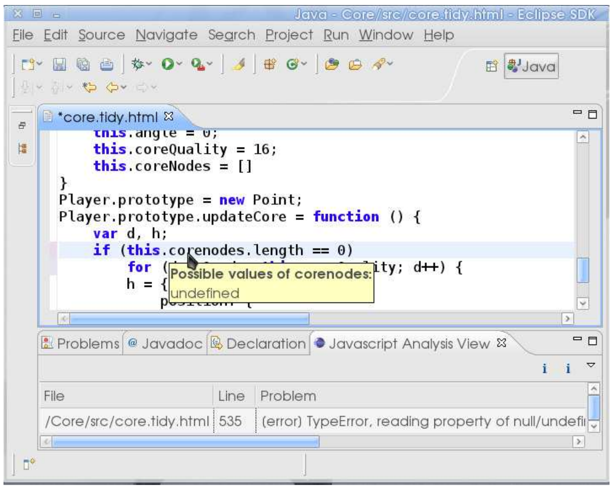
2. FINDING ERRORS WITH DATAFLOW ANALYSIS

The TAJS analysis tool infers an abstract state for each program point in a given JavaScript web application. Such an abstract state soundly models the possible states that may appear at runtime and can be used for detecting type-related errors and dead code. These errors often arise from wrong function parameters, misunderstandings of the runtime type coercion rules, or simple typos that can be tedious to find using testing.

We have approached the development of TAJS in stages. First, our focus has been on the abstract domain and dataflow constraints that are required for a sound and reasonably precise modeling of the basic operations of the JavaScript language itself and the native objects that are specified in the ECMAScript standard [3]. This involves an extraordinarily elaborate lattice structure and models of the intricate details of identifier and object property resolution, prototype chains, property attributes, scope chains, type coercions, etc. [7]. The resulting static analysis is flow- and partially context-sensitive. It performs constant propagation for primitive values and models object references using recency abstraction. For every expression, the analysis provides an over-approximation of its possible runtime types and values, which can be analyzed subsequently to detect likely errors.

Next, to reason about web application code, we also need to model the browser API, including the HTML DOM and the event system, with involves additional hundreds of objects, functions, and properties [6]. In parallel, we have developed new techniques for interprocedural dataflow analysis to boost performance. Our *lazy propagation* technique is particularly suitable for the large abstract states that we encounter [8]. More recently, we have taken the first step of handling common patterns of code that is dynamically generated using the `eval` function [5], using the study by Richards et al. [9] as a starting point.

Altogether, these techniques enable analysis of JavaScript web applications up to a few thousand lines of code, although the scalability is naturally highly affected by the complexity of the code. We have demonstrated that the approach can infer type information and call graphs with good precision


Java - Core/src/core.tidy.html - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
*core.tidy.html
Problems
Javadoc
Declaration
Javascript Analysis View

```
    this.angle = 0;
    this.coreQuality = 16;
    this.coreNodes = []
}
Player.prototype = new Point;
Player.prototype.updateCore = function () {
    var d, h;
    if (this.corenodes.length == 0)
        for (                    ity; d++) {
        h = {
            position: {
```

Possible values of corenodes:
undefined

File	Line	Problem
/Core/src/core.tidy.html	535	(error) TypeError, reading property of null/undefi

Figure 1: The TAJS analysis plug-in for Eclipse, reporting a programming error and highlighting the type inferred for the selected expression [6].

and provide useful warning messages when type-related errors occur. We envision such information being made available to the programmer during development; a screenshot from our prototype plugin for Eclipse is shown in Figure 1.

Our current work focuses on improving the analysis performance. As the average JavaScript programs become larger and often involve libraries, it becomes increasingly important that the scalability of the analysis is improved. Specifically, we are studying the performance bottlenecks that appear with applications that use jQuery, using the idea of correlation tracking that has recently been proposed by Sridharan et al. [11].

3. TOOL-SUPPORTED REFACTORING

Refactoring is a popular technique for improving the structure of programs while preserving their behavior. Tool support is indispensable for finding the necessary changes when the programmer suggests a specific refactoring and for ensuring that the program behavior is preserved. However, refactoring tools for JavaScript cannot use the techniques that have been developed for e.g. Java since they rely on information about static types and class hierarchies. As an example, no existing mainstream JavaScript IDE can perform even apparently simple refactorings, such as, renaming an object property, in a sound and precise manner.

In the JSRefactor project, we explore the use of pointer analysis as a foundation for providing better tool support for refactoring for JavaScript programs [4]. As a starting point we consider renaming of variables or object properties, but also more JavaScript-specific refactorings – encapsulation of properties and extraction of modules – that target programming idioms advocated by influential practitioners [2].

Beside supporting additional refactorings in our framework, an important next step is to improve the scalability of the underlying pointer analysis. On the theoretical side, it remains an interesting challenge how to ensure that the refactoring specifications we provide are sound with respect to the semantics of JavaScript.

4. AUTOMATED TESTING

Testing JavaScript web applications is tedious but necessary. The goal of the Artemis project is to automate the production of high-coverage test inputs [1]. This can be seen as a complementary approach to TAJS. Although testing cannot show absence of errors – in contrast to the static analysis approach we use in TAJS – one may argue that dynamic approaches to error detection are better suited for dynamic languages like JavaScript. As a case in point, `eval` causes no complications in Artemis, unlike in TAJS.

The approach we take in Artemis is to apply light-weight feedback-directed random testing. A test input consists of a sequence of parameterized events that trigger execution of code. The Artemis tool monitors the execution to collect information that suggests promising new inputs that may improve coverage.

Our first version of Artemis was based on Envjs, which is a simulated browser environment written in JavaScript, and included various heuristics for generating and prioritizing new inputs. We are currently integrating our algorithms into the more robust WebKit infrastructure and exploring more powerful heuristics for providing higher and faster coverage of typical JavaScript applications.

Acknowledgments

We appreciate the contributions to the TAJS, JSRefactor, and Artemis projects by Kristoffer Just Andersen, Shay Artzi, Julian Dolby, Matthias Diehn Ingesman, Jacob H.C. Kragh, Todd Millstein, Max Schäfer, Peter Thiemann, and Frank Tip. This work was supported by Google, IBM, and The Danish Research Council for Technology and Production.

5. REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE'11*, May 2011.
[2] D. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.
[3] ECMA. ECMAScript Language Specification, 3rd edition, 2000. ECMA-262.
[4] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *OOPSLA'11*, October 2011.
[5] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *ISSTA'12*, July 2012.
[6] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *ESEC/FSE'11*, September 2011.
[7] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS'09*, August 2009.
[8] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *SAS'10*, September 2010.
[9] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *ECOOP'11*, July 2011.
[10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of Javascript programs. In *PLDI'10*, June 2010.
[11] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP'12*, June 2012.

Big Bang

Designing a Statically-Typed Scripting Language

Pottayil Harisanker Menon Zachary Palmer Alexander Rozenshteyn Scott Smith
The Johns Hopkins University
{pharisa2, zachary.palmer, scott, arozens1}@jhu.edu

Overview

Scripting languages such as Python, Javascript, and Ruby are here to stay: they are terse, flexible, easy to learn, and can be used to quickly deploy small to medium-sized applications. However, scripting language programs run slowly [sho] and are harder to understand and debug since type information is not available at compile time. In the last twenty years, several projects have added static type systems to existing scripting languages [FAFH09, GJ90, BG93, Age95, FFK+96, THF10], but this technique has had limited success. The fundamental problem is that scripting language designs incorporate a number of decisions made without regard for static typing; adding typing or engineering optimizations retroactively without breaking compatibility is challenging. We believe that by starting fresh and designing a new statically-typed language, a cleaner system for "scripting-style" programming can be engineered.

This is a position paper outlining Big Bang, a new statically-typed scripting language. Static typing is feasible because we design the language and type system around a new, highly-flexible record-like data structure that we call the *onion*. Onions aim to unify imperative, object-oriented, and functional programming patterns without making artificial distinctions between these language paradigms. A subtype constraint inference type system is used [AW93, Hei94, EST95b, WS01], with improvements added to increase expressiveness, and to make the system more intuitively understandable for programmers.

The remainder of this paper describes the onion data combinator, the process of typechecking Big Bang, and some practical considerations involved in its implementation.

Onion-Oriented Programming

At the core of Big Bang is the extremely flexible *onion* data combinator. We introduce onions with some simple examples.

At first glance, onions often look like extensible records: `‘name "Sue" & ‘age 27 & ‘height 68` is an onion which simply combines labeled data items. We call the & operator action *onioning*, the combination of data. The only other non-primitive data constructor needed (beyond arrays) is the ability to *label* data (e.g., `‘age 27`). The combination of onions, labels, and functions allows all other data structures to be succinctly expressed. Onion concatenation, &, is a left-associative operator which gives rightmost precedence; `‘with 4 & ‘with 5 & ‘and 10` is equivalent to `‘with 5 & ‘and 10` since the `‘with 4` has been overridden.

In Big Bang, every datum is an onion: labeled data (e.g. `‘age 27`) is a 1-ary onion and is how a record of one field would be represented. But unlike records, labels are not required on data placed in an onion. 5 can be viewed as a 1-ary onion of type `int`; `5 & ‘with 4` and `"Two" & 2` are also onions. Operators that have a sensible meaning simply work: for example, `(5 & ‘with 4) + 2` returns 7 since addition implicitly projects the integer from the onion. The case expression is the only explicit data destructor; for example, `case (5 & ‘with 4) in { ‘with x -> x + 7 }` evaluates to 11 because x is bound to the contents of the `‘with` label in the onion. We use `(5 & ‘with 4).with + 7` as sugar for a single-branch case expression. case is also used for typecasing; `case x of { int -> 4; unit -> 5 }` evaluates to 4 if x is an int and 5 if x is a unit.

The underlying labeled data is mutable, but at the top level onions are *immutable*; this key restriction enables flexible subtyping. So, we can assign to x in the above examples, but we cannot change or remove the `‘with` from the onion in the case expression. This is in contrast with modern scripting languages in which object extension is accomplished by mutation. New onions can, however, be constructed by functional extension. For instance, consider the following Big Bang code:

```
def o = ‘x 3  in  def o’ = o & ‘y 5  in  o’.x + o’.y
```

Here, o contains only x while o’ contains both x and y.

Objects as onions Onions are additionally *self-aware* in the manner of primitive objects [AC96]. Objects are therefore easily encoded as onions. For example,

```
def point = ‘x 0 & ‘y 0 &
            ‘isZero λ_. (self.x == 0 and self.y == 0)
```

defines a Big Bang point object: the keyword `self` in a function in an onion refers to the onion enclosing that function.

Object extension can be modeled through onion extension; this allows the trivial definition of a mixin object. For instance,

```
def magMixin = ‘magnitude (λ_. self.x + self.y) in
def mpoint = point & magMixin  in  mpoint.magnitude ()
```

would typecheck correctly and evaluate to `0`. `self` is late bound as is traditional in inheritance and so the `self` in `magMixin` will be all of `mpoint` when the method is invoked.

Other programming constructs can also be expressed succinctly with onions. Classes, for instance, are simply syntactic sugar for objects which contain factory methods for other objects. Both single and multiple inheritance are modeled simply as object extension. We also plan to construct modules from onions, giving Big Bang a simple, lightweight module system.

Typing Big Bang

The Big Bang type system must be extremely expressive to capture the flexibility of onions and of duck typing. To meet this requirement, we start with a polymorphic subtype constraint-based type system and add several novel extensions to improve expressiveness, usability, and efficiency. The type system is entirely inference-driven; users are never required to write type annotations or look at particularly confusing types.

One improvement to existing constraint systems is how onion concatenation can be flexibly typed – any two onions can be concatenated and it is fully tracked by the type system. Existing works on record concatenation [AWL94, Hei94, Pot00] focus on symmetric concatenation which requires complex "field absence" information, destroying desirable monotonicity properties and increasing complexity. Concretely, we conjecture the monomorphic variant of our inference algorithm is polynomial, whereas the best known algorithm for concatenation with subtyping is NP-complete [PZ04, MW05]. We take a right precedence approach to the case of overlap simply because it is the way modern languages work: subclasses can override methods inherited from the superclass. This also resolves the multiple inheritance diamond problem in the manner of *e.g.* Python and Scala by making it asymmetric. Despite keeping only positive type information, we can also type an onion subtraction operation: Big Bang syntax `(‘with 4 & ‘and 5) &- ‘and` is typeable and returns `‘with 4`, removing the `‘and` label.

In Big Bang, *every* function is inferred a polymorphic type (following [WS01, LS08, KLS10], work in turn inspired by [Shi91, Age95]). Polymorphic function types are then instantiated at the application site. This is done *globally*, so every potential use of a function is taken into account. The key question in such an approach is when to stop generating fresh instantiations for the universal quantifier; in face of recursion, the naïve algorithm will not terminate. Consider the following:

```
(‘f λn. if n-1 = 0 then 0 else self.f (n-1 & ‘z n)).f 10
```

Note that `self` in the function body refers to the full 1-ary onion containing the label `‘f`; thus, the call to `self.f` is recursive. This toy example returns 0 at runtime, but it is called with ten different type parameters: `int`; `int & ‘z int`; `int & ‘z (int & ‘z int)`; and so on. This is termed a *polymorphically recursive* function. A standard solution to dealing with such unbounded cases in program analyses is to simply chop them off at some fixed point; nCFA is an early example of such an arbitrary cutoff [Shi91]. While arbitrary cutoffs may work for program analyses, they make type systems hard for users to understand and potentially brittle to small refactorings. For Big Bang we have developed a method extending [LS08, KLS10] which discovers and naturally merges exactly and only these recursive contours; there is no fixed bound n.

Lastly, we have developed *case constraints*, a new form of conditional type constraints, to accurately follow case branches when the parameter is statically known; this leads to more precise and more efficient typing. Case constraints are an extension of constraint types [Hei94, AWL94, Pot00] but are also path-sensitive w.r.t. side-effects in case branches. It is well known that polymorphic subtype constraint systems naturally encode positive union types via multiple lower bounds; negative union types are easily encoded by these case constraints.

Gradual tracking in Big Bang The Big Bang type system is, of course, a conservative approximation and will sometimes produce false positives. In these cases, a programmer should add explicit dynamic tracking. Unlike *gradual typing*, which starts with dynamic tags on all data and removes tags wherever possible, *gradual tracking* starts with no dynamic information and permits the programmer to incrementally add dynamic tags as necessary. For example, given a piece of code recursively iterating over the Big Bang list `[1,(),2,(),3,()]`, the type system may not statically know that, e.g., odd elements are always `int`s. A Big Bang programmer can still effectively use this list in two ways, depending the list's invariant. If the list simply contains values which are either integers or units, a case expression can be used to typecase on each element. But if the list always contains an integer followed by a unit and the programmer iterates over two elements at a time, the `int`/`unit` alternation will be statically inferred due to the particularly precise nature of our polyvariant inference algorithm.

The Big Bang type system is also capable of internally representing what is traditionally considered dynamic type information. For example, consider annotating strings to indicate that they are *safe* (such as is done by Django for HTML sanitization) by for example writing `"big" & ‘safe()`. Any use of that onion as a string will implicitly project the string value; that is, `concat ("big" & ‘safe()) "bang"` will evaluate to `"bigbang"`. We also expect concatenation to handle two safe strings properly; that is, `safeConcat ("big" & ‘safe()) ("bang" & ‘safe())` computes to `"bigbang" & ‘safe()`. The `safeConcat` function can check if a string is safe by using a case expression with a `‘safe x` pattern.

Helping programmers understand types Unfortunately, constraint sets produced by polymorphic subtype constraint-based type systems are difficult to read and understand; attempts to simplify the constraints [EST95a, Pot01] or to graphically render inferred constraints [FFK^{+}96] have met with only limited success. We believe these approaches do not abstract enough information from the underlying constraint sets. To show the programmer what type of data could be in a variable ob, we provide a shallow view of its possible onion(s); if ob is a method parameter which is passed either a `point` or `mpoint` (defined above), we show the *top-level slice* of the set of disjuncts: `{(‘x & ‘y & ‘isZero), (‘x & ‘y & ‘isZero & ‘magnitude)}`. Programmers are then free to interactively "drill in" to see deeper type structure when needed. Likewise, type errors are explained interactively; the compiler presents an error (e.g., "function cannot accept argument of type `int`"), the programmer asks for further information about the reasoning (either "show why that function cannot accept an `int`" or "show how the argument could be an `int`"), the compiler responds, and so forth.

Whole program typechecking Because programmers do not write type annotations in Big Bang, software modules cannot be coded to a type interface alone. But coding to a type interface is a shallow notion; many aspects of runtime behavior cannot be decidably encoded in a type system. Instead of relying on module boundaries, Big Bang uses a whole-program typechecking model. This does imply limitations on separate compilation of modules, although some analysis can still be done in isolation. Also, type errors will not be caught if no code activates the program flow on which the type error is found. But complete unit test coverage is critical in modern software development and unit tests activate these code paths. A Big Bang testing tool can statically verify complete unit test code coverage by checking for unused type constraints (which imply untested code). This way, code which has not been fully tested will generate type safety warnings.

Implementing Big Bang To test the Big Bang language design, we have implemented a typechecker and interpreter in Haskell. We are now starting on a full compiler implementation using the LLVM toolchain [LA04]. One particularly challenging task in compiling Big Bang is the optimization of memory layout; we must avoid runtime hashing to compute method offsets, but the flexibility and incremental construction of onions makes the static layout problem complex. We intend to build upon previous work in the area of flexible structure compilation [Oho95, WDMT02].

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Age95] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, 1995.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[AWL94] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[BG93] Gilad Bracha and David Griswold. Strongtalk: type-checking smalltalk in a production environment. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 215–230, New York, NY, USA, 1993. ACM.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, volume 30(10), pages 169–184, 1995.

[EST95b] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[FFK$^+$96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 23–32, New York, NY, USA, 1996. ACM.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for smalltalk. In *In Seventeenth Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, 1990.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 306–317, New York, NY, USA, 1994. ACM.

[KLS10] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 671–690, New York, NY, USA, 2010. ACM.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[LS08] Y. D. Liu and S. Smith. Pedigree types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

[MW05] Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 156–167, New York, NY, USA, 2005. ACM.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, November 1995.

[Pot00] François Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 320–335. Springer Verlag, March 2000.

[Pot01] François Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170:153–183, November 2001.

[PZ04] Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189(1):54 – 86, 2004.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.

[sho] shootout.debian.org. The computer language benchmarks game. `http://shootout.alioth.debian.org/`.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

[WDMT02] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 200, 2002.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, pages 99–117, 2001.

Gradual Specifications for Dimension Checking, Security, and Access Control Position Paper

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract

The essence of script to program evolution is that programs are equipped with specifications after the fact. Most work up to now has considered typing and verification aspects in these specifications. We believe that other aspects equally make sense and demonstrate this view with a discussion of gradual specifications for dimension checking and for security, in particular access control.

1. Introduction

Design by Contract (DBC) [8, 9] is the foundation for the construction of correct and secure software. DBC relies on contracts that determine the input/output behavior of a software component. This behavior can be statically verified as well as dynamically checked by run-time monitoring.

Sometimes the initial development of a software system takes place without proper attention to design. For example, a prototype may be created in a scripting language without a properly designed architecture in mind and without carefully crafted component interfaces. While such a state of affairs is acceptable for a prototype, it is often the case that such a prototype is further developed to a final product. However, due to the lack of well-designed interfaces, it is hard to establish correctness guarantees for such a system after the fact. Dynamically checked contracts are a proven means to augment such systems with correctness guarantees (e.g., robustness with respect to type errors) and gradual types enable the stepwise adoption of typing to growing parts of the system.

An entirely analogous scenario arises when software is developed according to an incomplete requirements specification. It is not unusual for customers to come up with additional requirements after the development team has converged on an overall design. For example, customers may suddenly realize that dimension checking, integrity and confidentiality, or access control are important for certain parts of their system and demand the adoption of suitable requirements.

Adherence to standard software engineering practices (even the very general caveat to "anticipate change") does not save the day if such cross-cutting requirements are newly imposed. Even a carefully designed system may require restructuring to have it successfully checked by a static analysis or by a suitable type system. For such cases, a gradual and evolutionary approach is appropriate. For each of the above-mentioned properties, we envision contracts that establish run-time guarantees in a way that can be integrated with existing static approaches. The considerate placement of contracts can be employed to immediately guarantee the desired properties using dynamic enforcement (monitoring) at run time. Afterwards, the performance and the robustness of the system can be improved by creating statically checked "islands" in the code that need not be dynamically monitored. Given enough resources, these islands can be gradually extended to encompass the entire program.

Dimension checking Dimension types [7] guarantee that the code does not contain computations that mix up different physical dimensions. They ensure that, for example, areas and lengths are not confused or that imperial and metric units are not improperly mixed up.

A suitable gradual system would augment each value with a run-time dimension annotation and instrument the arithmetic operations to propagate these annotations accordingly. The conversions into a checked island of the program would strip away the annotations so that the island can employ plain arithmetic, whereas the conversions out of the island wrap numbers with dimension annotations, again.

Integrity and confidentiality Information flow can be encoded in a type system [12] and statically avoid breaches of confidentiality. Integrity checks can be added by augmenting the type system in the style demonstrated by the SLam calculus [6].

Disney and Flanagan have proposed a simple type system for gradual checking of information flow [1], thus giving evidence for the feasibility of the approach. However, their system appears to be less expressive than the SLam calculus.

Access control Java enforces access control policies by performing stack inspection at run time. Pottier and coworkers [10] show that a suitable type system can statically guarantee most of these properties without the associated run-time cost. However, their system cannot model fine-grained policies like disabling file access based on the particular path name.

In this case, the dynamic enforcement and the type system both already exist. A gradual approach would be interesting because it would make it possible to exploit the additional expressiveness of the run-time enforcement where needed and to enjoy the improved efficiency elsewhere.

Even in a setting where software is developed with contracts from the beginning and where the cross-cutting requirements are known from the start, it can make sense to start out with dynamically checked contracts. As a dynamic contract language can often express properties that are out of reach of static checking, such an approach can ensure that the initially stated contracts are as precise as possible.

In both cases it is often desirable to integrate static checking or to migrate parts of the system to stating checking. The motivation for a migration may include concerns for efficiency and robustness. An integration may also also be required for stating liveness properties or, as a prominent example from the security realm, noninterference, which is known not to be amenable to pure dynamic checking [4].

2. A Gradual Approach to Access Control

For contracts based on types, gradual typing [11] provides a migration path from dynamically checked properties to a statically verified system. Drawing the analogy, a gradual access control type system provides a migration path from dynamically enforced access control to statically checked access control. In this case, the type conversions can also serve as boundaries to separate untrusted, dynamically checked parts of a program from trusted, statically checked parts. In particular, the dynamically checked parts could be could generated or downloaded at run time.

We are currently working on such a migration path for a range of security properties. Besides the integrity and confidentiality properties, which are captured by standard multi-level security type systems [12], we are interested in access control on the object level. This access control differs significantly from the resource control of Pottier and coworkers[10]. While their system controls access to resources like the audio subsystem, the file system, or network communication, our envisaged system is geared at restricting the objects accessible to a component using path expressions.

In prior work [5], we introduced a contract system of dynamically enforced access permissions with a path-based semantics which fit well in a verification scenario. To complete the migration path for verification, a corresponding static access permission (effect) system is required. To address the security scenario in a satisfactory way, a different, location-based semantics for dynamic checking is required along with the corresponding static system. In this way, we create a methodology for gradual verification and security hardening of software, in particular for scripting languages, which facilitates a gradual migration from dynamically checked to statically verified components.

As a bonus to a purely static system, a gradual system enables us to include manifestation and focusing where a dynamic check establishes a property which is then carried through in the static system. These ideas have been pioneered for types [2, 3] and they seamlessly extend to a setting which includes security properties.

3. Gradual Access Control for References

To investigate the design space of a gradual system for access control, we construct a calculus that performs access control for references with the following syntax.

$$\begin{array}{llll} \textit{Mode} & m & ::= & \mathbf{1} \mid \mathbf{0} \\ \textit{Exp} & e & ::= & x \mid \lambda^m x.e \mid e\ e \mid \mathsf{ref}\ a\ e \mid !e \mid e := e \\ & & \mid & \emptyset \mid \mathsf{loc}\ e \mid \mathsf{like}\ e \mid e \cup e \\ & & \mid & \mathsf{restrict}\ e\ \mathsf{in}\ e \mid \mathsf{permit}\ e\ \mathsf{in}\ e \end{array}$$

The base calculus is a call-by-value lambda calculus with ML-style references. The a annotation of the operator for creating references is a marker for the allocation sites in a program and it is drawn from some unspecified set. These markers are usually unique, but that is not a requirement. We explain the m annotation of the lambda at the end of this subsection.

The next line of the expression grammar specifies syntax for constructing designators for sets of references. $\emptyset$ stands for the empty set, $\mathsf{loc}\ e$ expects that e evaluates to a reference ℓ and then stands for the singleton set $\{\ell\}$. The expression $\mathsf{like}\ e$ also expects that e evaluates to a reference ℓ, but then it stands for the set of all references that are allocated with the same allocation site marker as ℓ. The join operator $\cup$ just builds the union of two such sets.

The restrict and permit operators in the last line both accept as their first argument a descriptor of a set of references and restrict the access rights for the evaluation of their second argument. Writing $\mathsf{restrict}\ e_0\ \mathsf{in}\ e$ disallows access to the references designated by e_0 scoped over e whereas $\mathsf{permit}\ e_0\ \mathsf{in}\ e$ disallows access to all references **not** designated by e_0 scoped over e.

With "scoped over" we mean that the restriction acts like a wrapper on e. It recursively restricts the execution below e and all computations that are executed on behalf of e. That is, if e returns a function, the body of this function is also restricted, and so on.

If e is a higher-order function and a function g is passed as an argument, the programmer may choose under which restriction this function executes. If g is an overriding lambda marked with $m = \mathbf{0}$, then it runs with the restriction of its creation site. Otherwise it inherits the restrictions of its caller and of its creation site.

4. Conclusion

This document is a preliminary investigation of the issues that arise in a gradual type system for access control. We first argue that such a system (along with some related systems) makes sense from the general point of view of software engineering. Then we exhibit a first design of a suitable calculus that only considers references instead of objects and access paths.

We are currently working on a type system for a gradual extension of this calculus. There are still a number of open questions concerning the correct notion of subtyping and the annotation overhead. An embedding transformation from a simply-typed lambda calculus into our system would alleviate that overhead. We have yet to establish formal properties like type safety and, potentially, a suitable variation of a blame theorem (in the style of Wadler and Findler [13]).

References

[1] T. Disney and C. Flanagan. Gradual information flow typing. In *STOP 2011*, 2011.

[2] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. 2002 PLDI*, pages 13–24, Berlin, Germany, June 2002. ACM Press.

[3] C. Flanagan. Hybrid type checking. In S. Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 245–256, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

[4] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

[5] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *Proc. 39th ACM Symp. POPL*, pages 111–122, Philadelphia, USA, Jan. 2012. ACM Press.

[6] N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In L. Cardelli, editor, *Proc. 25th ACM Symp. POPL*, pages 365–377, San Diego, CA, USA, Jan. 1998. ACM Press.

[7] A. Kennedy. Dimension types. In D. Sannella, editor, *Proc. 5th ESOP*, volume 788 of *LNCS*, pages 348–362, Edinburgh, UK, Apr. 1994. Springer.

[8] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[10] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM TOPLAS*, 27(2):344–382, 2005.

[11] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

[12] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[13] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In G. Castagna, editor, *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer-Verlag.

Towards Gradual Typing in Jython

Michael M. Vitousek Shashank Bharadwaj Jeremy G. Siek
University of Colorado at Boulder
{michael.vitousek, shashank.bharadwaj, jeremy.siek}@colorado.edu

1. Introduction

The Jython implementation of Python for the JVM [1] stands to benefit greatly from the introduction of gradual typing. In particular, it may lead to improved program efficiency, static detection of type errors, and improved modularity of runtime error detection through blame tracking [2]. However, there are tensions between these goals. For example, the addition of type annotations to a program often causes the compiler to insert casts to mediate between static and dynamically typed code and these casts incur runtime overhead [3, 7].

Researchers have demonstrated space efficient blame tracking for casts involving first-class functions but the technique required a second value-form at function type, a casted function [4, 5]. The consequence for compilation is that generated code for function application must perform a dispatch and thereby incurs overhead, even for function applications residing in statically typed code. At Dagstuhl in January, Siek suggested a solution to this problem by storing the "threesome" as part of the closure representation and by moving the work of casting from the caller to the callee. In Section 2 we elaborate on this solution.

After functions, the next major challenge for efficient gradual typing in Python is how to efficiently implement casts involving objects. Objects are problematic because Python, as an imperative and dynamically typed language, allows *strong updates*, that is, changes to objects that affect their type. This problem has appeared in other guises, such as covariant arrays in Java and changes in typestate [6]. The approaches to date either guard every read access by a runtime type test or require reference counting. Both approaches impose significant runtime overhead, even for statically typed code. In Section 3 we propose an alternative in which we only guard strong updates and only allow the type of an object to change in a monotonically decreasing fashion with respect to naïve subtyping.

2. Function casts

We begin by considering the implementation of function casts. The naïve approach would be to have every cast on a function create a new function that "wraps" up the function, but this method is not scalable to situations where a function is repeatedly passed from static code to dynamic code and vice versa. Consider the following partially-typed Gradual Jython code, which traverses a file directory tree, applies a function *fun* to each file in the structure, and prints its output:

```
def explore_files(files, fun):
  for file in files:
    if file.is_directory():
      explore_dir(file, fun)
    else: print fun(file)
def explore_dir(dir:file, fun:file → str) → unit:
  explore_files(file.members(), fun)
```

Because a new wrapper is added around *fun* every time it passed through from one function to another, there is a $O(n)$ space blowup, rendering this approach infeasible.

Siek and Wadler developed a partial solution to this issue by attaching a "threesome" to a casted function and by merging threesomes when a casted function is cast yet again [4]. While this eliminates the $O(n)$ blowup of function wrapping, it also increases the overhead of function calls, because the compiled call site needs to check if the value being called is a threesome-wrapped function or a bare function. As such, the rule for compiling functions becomes

$$\begin{array}{l} [\![e_1(e_2)]\!] = \\ \quad \mathsf{let}\ f = [\![e_1]\!]\ \mathsf{in} \\ \quad \mathsf{case}\ f\ \mathsf{of} \\ \quad\ \mid \mathsf{Casted}\ f'\,\mathcal{K} \Rightarrow f'([\![e_2]\!] : dom(\mathcal{K})) : cod(\mathcal{K}) \\ \quad\ \mid \mathsf{Function}\ f' \Rightarrow f'.fun(f'.fvs, [\![e_2]\!]) \end{array}$$

We see a path to solving this problem by using a combination of the naïve wrapping approach and the threesomes approach. In this formulation, function closures contain pointers to a stored threesome, in addition to the typical function code and values for free variables. In a closure that has not been cast, the threesome is an identity cast. Applying a cast to a closure for the first time installs a generic wrapper function that performs casting on the argument, calls the original function, and casts the return value. Additional casts applied to the closure simply alter the threesome.

With this modification to function closures, call sites are returned to their simple form, with the exception that we pass the entire closure into the function instead of just its free variables:

$$[\![e_1(e_2)]\!] = \mathsf{let}\ f = [\![e_1]\!]\ \mathsf{in}\ f.fun(f, [\![e_2]\!])$$

The wrapper installed onto casted function bodies accesses the closure's threesome to cast the function from its original to its final type. Uncasted functions, lacking wrappers, simply ignore the threesome. To enable this treatment of threesomes as data, we depart from previous work and make threesomes into first-class entities.

We believe this approach to function casts will suffice to eliminate overhead at the call site of uncasted functions while continuing to support blame tracking to produce useful error messages.

3. Object casts

Our implementation of function casts relies on the fact that Python functions are immutable — a property that does not hold for Python objects. This complicates the design of gradual object casts in Jython, requiring a different approach from that used for functions.[1]

3.1 Motivation

The complicating effects of imperative update on gradual typing are reflected in the following code. This program constructs an object

[1] Python does not have mutable reference cells, but a language with *ref*s would have to consider similar issues.

obj including a member x with value 10 and calls *get_ref* which returns a function that is holding a reference to obj. The reference to obj relies on member x having type int. Upon returning from *get_ref*, the program mutates $obj.x$ to be a string value and then calls *x_ref*, which tries to reference member x as an integer.

```
1: $obj$:dyn = $\{x = 10, y = \mathsf{True}\}$ #Object initialization
2: def $get\_ref$(obj:$\{x$:int, $y$:dyn$\}$) $\rightarrow$ (unit $\rightarrow$ int):
3:     return $\lambda t$:unit. $obj.x$  #Capture typed reference
4: $x\_ref$:(unit $\rightarrow$ int) = $get\_ref(obj)$
5: $obj.x$ = "Hello!"
6: print ($x\_ref() + 10$)
```

This program should fail because $obj.x$ is updated to contain a string but has been casted to int. However, we would like to detect this error without incurring too much overhead on member accesses, and while still being able to blame the code ultimately responsible for violating the constraints on the types. This choice has ramifications for the internal representation of Jython objects.

3.2 Approaches to object casts

We review the traditional approach that relies on access checking and then present our new approach to object casts.

3.2.1 Access checking

One solution to this problem is to use additional inserted casts to check that object members conform to their expected types when accessed. This approach would require a cast or runtime check to be inserted at the access $obj.x$ at line 3 above. In this case, $obj.x$ would be cast to int — and in this particular program, the cast would fail, since at the time that x is finally dereferenced (at line 6) its value is a str.

This approach maintains the "Pythonic" flexibility of object updates even when typed variables view objects. On the other hand, casting members at their access sites adds overhead, and free field mutation may make it difficult to use an object representation conducive to fast field access. While we believe we can use JVM techniques such as InvokeDynamic or MethodHandles to minimize this overhead, we have the additional problem that the code point blamed is the access site (line 3), not the location of the update that invalidated the access' assumptions (line 5).

3.2.2 Monotonic objects

A second method for performing function casts involves permanently restricting the types of object members when casts are applied. This approach, which we call "monotonic objects", requires that the object itself record the most specific type (the *meet* with respect to naïve subtyping [4]) that its members have been viewed through. Successful casts update the meet as needed. This system detects the error in the above example at the update — when *obj* is passed to *get_ref*, the x field of *obj* is forever after restricted to containing ints, and so the update at line 5 fails and is blamed.

This approach enables a representation for objects that enable fast field lookup from static code. Objects consist of a dictionary, required for dynamic accesses, and an array of values and their meet types, as shown in Figure 1. When another, differently-typed reference is made to an object, its meet types mutate.

Wolff et al. [6] offer an alternative approach in which strong updates are allowed so long as they respect the types of all current references to the object. Their approach requires a form of reference counting that induces significant runtime overhead and it makes the semantics of the program depend on the speed of the garbage collector. Our monotonic object approach provides a more efficient, but in some ways less flexible, alternative.

We plan to further investigate monotonic objects and access checking. The monotonic approach promises more efficient field

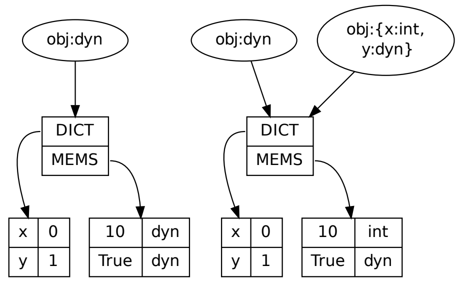


Figure 1. Representations of *obj* at line 1 and after line 4.

accesses, but the flow-sensitive restrictions it places on object values may be problematic in practice, and alternatively we may be able to reduce the runtime overhead of access checking.

4. Conclusions

Handling casts on nontrivial program data is a critical challenge for implementing gradual typing in a language like Jython. We have identified several of the problems that need to be solved to make gradual typing feasible in Jython — specifically, correct and efficient function and object casts — and have laid out our current strategies for confronting these challenges. More work is required to determine the best approaches, but our work thus far seems promising and we are confident that these challenges can be solved.

References

[1] The Jython Project. URL `http://jython.org`.

[2] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.

[3] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[4] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL '10*, pages 365–376. ACM, 2010.

[5] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *ESOP '09*, pages 17–31. Springer, 2009.

[6] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 459–483, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2010. ACM.