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: typechecking 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.