Available online at www.sciencedirect.com
SciVerse ScienceDirect
Electronic Notes in Theoretical Computer Science 279 (1) (2011) 47–59
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Implementing a Language with Flow-Sensitive and Structural Typing on the JVM

David J. Pearce[1] James Noble[2]

School of Engineering and Computer Science
Victoria University of Wellington, NZ

Abstract

Dynamically typed languages are flexible and impose few burdens on the programmer. In contrast, static typing leads to software that is more efficient and has fewer errors. However, static type systems traditionally require every variable to have one type, and that relationships between types (e.g. subclassing) be declared explicitly.
The Whiley language aims to hit a sweet spot between dynamic and static typing. This is achieved through structural subtyping and by typing variables in a flow-sensitive fashion. Whiley compiles to the JVM, and this presents a number of challenges. In this paper, we discuss the implementation of Whiley's type system on the JVM.

Keywords: Flow-Sensitive, Structural Subtyping, Java, JVM

1 Introduction

Statically typed programming languages (e.g. Java, C#, C++, etc) lead to programs which are more efficient and have fewer errors [11,2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible which helps reduce overheads and increase productivity [37,47,34,7]. Furthermore, in recent times, there has been a significant shift towards dynamically typed languages [38].

Numerous attempts have been made to bridge the gap between static and dynamic languages. Scala [45], C#3.0 [6], OCaml [43] and, most recently, Java 7 all employ local type inference (in some form) to reduce syntactic overhead. Techniques such as gradual typing [46,50], soft typing [11] and hybrid typing [20] enable a transitory position where some parts of a program are statically typed, and others are

[1] Email: djp@ecs.vuw.ac.nz

[2] Email: kjx@ecs.vuw.ac.nz

doi:10.1016/j.entcs.2011.11.005

not. Alternatively, type inference can be used (in some situations) to reconstruct types "after the fact" for programs written in dynamic languages [3,23].

Whiley is a statically-typed language which, for the most part, has the look and feel of a dynamic language. This is achieved with an extremely flexible type system which utilises the following features:

- **Flow-sensitive types**, which are adopted from flow-sensitive program analysis (e.g. [22,30,14]) and allow variables to have different types at different points.
- **Structural Subtyping**, where subtyping between data types is implict and based on their structure.

Taken together, these offer several advantages over traditional nominal typing, where types are named and subtyping relationships explicitly declared.

1.1 Contributions

The contributions of this paper are:

(i) We discuss the flow-sensitive and structural typing system used in the Whiley language.

(ii) We detail our implementation of these features on the JVM, and identify a number of challenges.

An open source implementation of the Whiley language is freely available from `http://whiley.org`. Finally, a detailed formalisation of Whiley's type system, including some discussion of JVM implementation, can be found here [39].

2 Whiley

In this section, we present a series of examples showing Whiley's key features. In the following section, we'll discuss their implementaiton in the JVM.

2.1 Implicit Declaration

Most contemporary statically typed languages require variables be explicitly declared (FORTRAN is one exception here). Compared with dynamically typed languages, this is an extra burden for the programmer, particularly when a variable's type can be inferred from assigned expression(s). In Whiley, local variables are declared by assignment:

```
int average([int] items):
    v = 0
    for i in items:
        v = v + items[i]
    return v / |items|
```

Here, `items` is a list of `int`s, whilst `|items|` returns its length. The variable `v` is used to accumulate the sum of all elements in the list. Variable `v` is declared by the assignment from 0 and, since this has type `int`, `v` has type `int` after the assignment.

2.2 Union Types

Nullable references have proved a significant source of error in languages such as Java [29]. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references [40]. Many solutions have been proposed which strictly distinguish these two forms using static type systems [18,17,41,35].

Whiley's type system lends itself naturally to handling this problem because it supports *union types* (see e.g. [5,31]). These allow variables to hold values from different types, rather than just one type. The following illustrates:

```
null|int indexOf(string str, char c):
   ...

[string] split(string str, char c):
   idx = indexOf(str,c)
   if idx ~= int:
       // matched an occurrence
       below = str[0..idx]
       above = str[idx..]
       return [below,above]
   else:
       return [str] // no occurrence
```

Here, `indexOf()` returns the first index of a character in the string, or `null` if there is none. The type `null|int` is a union type, meaning it is either an `int` or `null`.

In the above example, Whiley's type system seamlessly ensures that `null` is never dereferenced. This is because the type `null|int` cannot be treated as an `int`. Instead, we must first perform a runtime type test to ensure it is an `int`. Whiley automatically retypes `idx` to `int` when this is known to be true, thereby avoiding any awkward and unnecessary syntax (e.g. a cast as required in [4,35]).

2.3 Flow-Sensitive Typing

The following code shows the definition of a simple hierarchy of Shapes in Whiley, and a function that returns the area of any of these three types of Shapes.

```
define Circle as {int x, int y, int radius}
define Square as {int x, int y, int dimension}
define Rectangle as {int x, int y,
                     int width, int height}

define Shape as Circle | Square | Rectangle

real area(Shape s):
    if s ~= Circle:
        return PI * s.radius * s.radius
    else if s ~= Square:
        return s.dimension * s.dimension
    else:
        return s.width * s.height
```

A `Shape` is a union type — either a `Circle`, `Square` or `Rectangle` (which are all themselves record types). The code employs a runtime type test, "`s ~= Circle`", to distinguish the different kinds of `Shapes`. This is similar to Java's `instanceof` or Eiffel's reverse assignment. Unlike Java, Whiley retypes `s` to be of type `Circle` on the true branch of the `if` statement, so there is no need to cast `s` explicitly to `Circle` before accessing the `Circle`-specific field `radius`. Similary, on the false branch, Whiley retypes `s` to the union type `Square|Rectangle`, and then to `Square` or `Rectangle` within the next `if`.

Implementing these Shapes in most statically-typed languages would be more cumbersome and more verbose. In modern object-oriented languages, like Java, expressions must still be *explicitly* retyped. For example, after a test such as `s instanceof Circle`, we must introduce a new variable, say `c`, with type `Circle` as an alias for `s`, and use `c` whenever we wanted to access `s` as a circle.

2.4 Structural Subtyping

Statically typed languages, such as Java, employ *nominal typing* for recursive data types. This results in rigid hierarchies which are often difficult to extend. In contrast, Whiley employs *structural subtyping* of records [10] to give greater flexibility. For example, the following code defines a `Border` record:

```
define Border as {int x, int y, int width, int height}
```

Any instance of `Border` has identical structure to an instance of `Rectangle`. Thus, wherever a `Border` is required, a `Rectangle` can be provided and vice-versa — even if the `Border` definition was written long after the `Rectangle`, and even though `Rectangle` makes no mention of `Border`.

The focus on structural, rather than nominal, types in Whiley is also evident in the way instances are created:

```
bool contains(int x, int y, Border b):
    return b.x <= x && x < (b.x + b.width) &&
           b.y <= y && y < (b.y + b.height)

bool example(int x, int y):
    rect = {x: 1, y: 2, width: 10, height: 3}
    return contains(x,y,rect)
```

Here, function `example()` creates a record instance with fields `x`, `y`, `width` and `height`, and assigns each an initial value. Despite not being associated with a name, such as `Border` or `Rectangle`, it can be freely passed into functions expecting such types, since they have identical *structure*.

2.5 Value Semantics

In Whiley, all compound structures (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, or most functional languages) — but unlike functional languages (and like Pascal) values of compound types can be updated in place.

Value semantics implies that updates to the value of a variable can only affect that variable, and the only way information can flow out of a procedure is through that procedure's return value. Furthermore, Whiley has no general, mutable heap comparable to those found in object-oriented languages. Consider the following:

```
int f([int] xs):
    ys = xs
    ys[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `ys` does not affect `xs`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, changes can only be communicated out of a function by explictly returning a value.

Whiley also provides strong guarantees regarding subtyping of primitive types (i.e. integers and reals). In Whiley, `int`s and `real`s represent unbounded integers and rationals, which ensures `int` $\leq$ `real` has true subset semantics (i.e. every `int` can be represented by a `real`). This is not true for e.g. Java, where there are `int` (resp. `long`) values which cannot be represented using `float` (resp. `double`) [26, §5.1.2].

2.6 Incremental Construction

A common pattern arises in statically typed languages when a structure is built up piecemeal. Usually, the pieces are stored in local variables until all are available and the structure can be finally created. In dynamic languages it is much more common to assign pieces to the structure as they are created and, thus, at any given point a partially complete version of the structure is available. This reduces syntactic overhead, and also exposes opportunities for code reuse. For example, the partial structure can be passed to functions that can operate on what is available. In languages like Java, doing this requires creating a separate (intermediate) object.

In Whiley, structures can also be constructed piecemeal. For example:

```
BinOp parseBinaryExpression():
    v = {} // empty record
    v.lhs = parseExpression()
    v.op = parseOperator()
    v.rhs = parseExpression()
    return v
```

After the first assignment, `v` is an empty record. Then, after the second it has type `{Expr lhs}`, after the third it has type `{Expr lhs, Op op}`, and after the fourth it has type `{Expr lhs,Expr rhs,Op op}`. This also illustrates the benefits of Whiley's value semantics with update: the value semantics ensure that there can never be any alias to the value contained in `v`; while updates permit `v` to be built imperatively, one update at a time.

2.7 *Structural Updates*

Static type systems normally require updates to compound types, such as list and records, to respect the element or field type in question. Whiley's value semantics also enables flexible updates to structures without the aliasing problems that typically arise in object-oriented languages. For example, assigning a `float` to an element of an `int` array is not permitted in Java. To work around this, programmers typically either clone the structure in question, or work around the type system using casting (or similar).

In Whiley, updates to lists and records are always permitted. For example:

```
define Point as {int x, int y}
define RealPoint as {real x, real y}

RealPoint normalise(Point p, int w, int h):
    p.x = p.x / w
    p.y = p.y / h
    return p
```

Here, the type of `p` is updated to `{real x,int y}` after `p.x` is assigned, and `{real x,real y}` after `p.y` is assigned. Similarly, for lists we could write:

```
[real] normalise([int] items, int max):
    for i in 0..|items|:
        items[i] = items[i] / max
    return items
```

Here, the type of `items` is updated to `[real]` by the assignment. Thus, Whiley's type system permits an in-place update from integer to real without requiring any explicit casts, or other type system abuses (e.g. exploiting raw types, such as `List`, in Java).

3 Implementation on the JVM

The Whiley language compiles down to Java bytecode, and runs on the JVM. In this section, we outline how Whiley's data types are represented on the JVM.

3.1 Numerics

Whiley represents numbers on the JVM in a similar fashion to Clojure [28]. More specifically, `int`s and `real`s are represented using custom `BigInteger` and `BigRational` classes which automatically resize to prevent overflow. Whiley requires that integers are truly treated as subtypes of reals. For example:

```
real f(int x):
    if x >= 10:
        x = 9.99
    return x
```

At the control-flow join after the `if` statement, `x` holds either an `int` or a `real`. Since `real` values are implemented as `BigRational`s on the JVM, we must coerce `x` from a `BigInteger` on the false branch.

3.2 Records

The implementation of Whiley's record types must enable structural subtyping. A simple approach (used in many dynamic languages), is to translate them as `HashMap`s which map field names to values. This ensures that record objects can be passed around freely, provided they have the required fields.

One issue with this approach, is that each field access requires a `HashMap` lookup which, although relatively fast, does not compare well with Java (where field accesses are constant time). Whiley's semantics enable more efficient implementations. In particular, the type `{int x}` is a record containing *exactly* one field `x`. Thus, records can have a static layout to give constant time access [40]. For example, records can be implemented on the JVM using arrays of references, rather than `HashMap`s. In this approach, every field corresponds to a slot in the array whose index is determined by a lexicographic ordering of fields.

To implement records using a static layout requires the compiler to insert coercions to support subtyping. Consider the following:

```
define R1 as {int x, int y}
define R2 as {int y}

R1 r1 = {x: 10, y: 20}
R2 r2 = r1
```

Let us assume our records are implemented using a static layout where fields are ordered alphabetically. Thus, in `R1`, field `x` occupies the first slot, and field `y` the second. Similarly, field `y` corresponds to the first slot of `R2` and, hence, `R1` is not compatible with `R2`. Instead, we must convert an instance of `R1` into an instance of `R2` by constructing a new array consisting of a single field, and populating that with the second slot (i.e. field `y`). This conversion is safe because of Whiley's value semantics — that is, two variables' values can never be aliased.

3.3 Collections

Whiley provides first-class lists, sets and maps which are translated on the JVM into `ArrayList`s, `HashSet`s and `HashMap`s respectively. Of course, all these collection types must have value semantics in Whiley. Recall that, in the following, updating `ys` does not update `xs`:

```
int f([int] xs):
    ys = xs
    ys[0] = 1
    ...
```

A naive translation of this code to the JVM would `clone()` the `ArrayList` referred to by `xs`, and assign this to `ys`. This can result in a lot of unnecessary copying of data and there are several simple strategies to reduce this cost:

(i) Use a `CopyOnWriteArrayList` to ensure that a full copy of the data is only made when it is actually necessary.

(ii) Use an intraprocedural dataflow analysis to determine when a variable is no longer used. For example, in the above, if `xs` is not live after the assignment to `ys` then cloning it is unnecessary.

(iii) Exploit compiler inferred `read-only` modifiers for function parameters. Such modifiers can be embedded into the JVM bytecode, and used to identify situations when an argument for an invocation does not need to be cloned.

Currently, we employ only `CopyOnWriteArrayList`s to improve performance, although we would like to further examine the benefits of those other approaches.

3.4 Runtime Type Tests

Implementing runtime type tests on the JVM represents something of a challenge. Whilst many runtime type tests translate directly using appropriate `instanceof` tests, this is not always the case:

```
int f(real x):
    if x ~= int:
        return x
    return 0
```

Although Whiley `int`s are implemented as Java `BigInteger`s, this test cannot be translated to "`e instanceof BigInteger`". This is because of Whiley's subtyping rules: `x` will be implemented by an instance of `BigRational` on entry, but because Whiley `int`s are subtypes of Whiley `real`s, the subtype check should succeed if the actual real passed in is actually an integer. The test is therefore translated into a check to see whether the given `BigRational` instance corresponds to an integer or not.

Union types also present a challenge because distinguishing the different cases is not always straightforward. For example:

```
define data as [real] | [[int]]

int f(data d):
    if d ∼= [[int]]:
        return |d[0]|
    return |d|
```

Since variable d is guaranteed to be a list of some sought, its type on entry is translated to List on the JVM. Thus, Whiley cannot use an instanceof test on d directly to distinguish the two cases. Instead, we must examine the first element of the list and test this. Thus, if the first element of the list is itself a list, then the true branch is taken[3].

The following example presents another interesting challenge:

```
int f([real] d):
    if d ∼= [int]:
        return d[0]
    return 0
```

To translate this test, we must loop over every element in the list and check whether or not it is an integer. Furthermore, if this is the case, we must convert the list into a list of BigInteger, rather than BigRational.

Finally, records present an interesting issue. Consider the following example:

```
define Rt as {int x, int y} | {int x, int z}

int unpackSecond(Rt r):
    if r ∼= {int x,int y}:
        return r.y
    return r.z
```

Since variable r is guaranteed to be a record of some sort, its type on entry is translated as Object[]. Thus, implementing the type test using instanceof does not make sense, as this will not distinguish the two different kinds of record. Instead, we must check whether r has fields x and y, or not. To support this, the representation of records must also associate the corresponding field name with each slot in the Object[] array. This is achieved by reserving the first slot of the array as a reference to an array of field names, each of which identifies the name of a remaining slot from the outer array.

Distinguishing between different record types can be optimised by reducing the number of field presence checks. For example, in the above, there is little point in checking for the presence of field x, since it is guaranteed in both cases. The Whiley compiler generates the minimal number of field checks to be certain which of the possible cases is present.

[3] Note that in the case of an empty list, then type test always holds

4 Related Work

In this section, we concentrate primarily on work relating to Whiley's flow-sensitive type system.

4.1 Dataflow Analysis

Flow-sensitive dataflow analysis has been used to infer various kinds of information, including: *flow-sensitive type qualifiers* [21,22,12,17,27,13,1,41,4,35], *information flow* [30,44,36], *typestates* [49,19,8], *bytecode verification* [33,32] and more.

Type qualifiers constrain the possible values a variable may hold. CQual is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/O [21,22]. CQual does not account for the effects of conditionals and, hence, retyping is impossible. The work of Chin *et al.* is similar, but flow-insensitive [12,13] JQual extended these systems to Java, and considered whole-program (flow-insensitive) inference [27]. AliasJava introduced several qualifiers for reasoning about object ownership [1]. The `unique` qualifier indicates a variable holds the only reference to an object; similarly, `owned` is used to confine an object to the scope of its enclosing "owner" object.

Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [18]. Here, variables are annotated with `NonNull` to indicate they cannot hold `null`. Non-null qualifiers are interesting because they require variables be retyped after conditionals (i.e. retyping `v` from `Nullable` to `NonNull` after `v!=null`). Fähndrich and Leino hint at the use of retyping, but focus primarily on issues related to object constructors. Ekman *et al.* implemented this system within the JustAdd compiler, although few details are given regarding variable retyping [17]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `!=null` checks [41]. The JACK tool is similar, but focuses on bytecode verification instead [35]. This extends the bytecode verifier with an extra level of indirection called *type aliasing*. This enables the system to retype a variable `x` as `@NonNull` in the body a `if(x!=null)` conditional. The algorithm is formalised using a flow-sensitive type system operating on Java bytecode. JavaCOP provides an expressive language for writing type system extensions, including non-null types [4]. This system is flow-insensitive and cannot account for the effects of conditionals; as a work around, the tool allows assignment from a nullable variable `x` to a non-null variable if this is the first statement after a `x!=null` conditional.

Information Flow Analysis is the problem is tracking the flow of information, usually to restrict certain flows based for security reasons. The work of Hunt and Sands is relevant here, since they adopt a flow-sensitive approach [30]. Their system is presented in the context of a simple While language not dissimilar to ours, although they do not account for the effect of conditionals. Russo *et al.* use an extended version of this system to compare dynamic and static approaches [44]. They demonstrate that a purely dynamic system will reject programs that are considered type-safe under the Hunt and Sands system. JFlow extends Java with statically checked flow annotations which are flow-insensitive [36]. Finally, Chugh *et al.* developed a constraint-based (flow-insensitive) information flow analysis of JavaScript [15].

Typestate Analysis focuses on flow-sensitive reasoning about the state of objects, normally to enforce temporal safety properties. Typestates are finite-state automatons which can encode usage rules for common APIs (e.g. a file is never read before being opened), and were pioneered by Strom and Yellin [48,49]. Fink *et al.* present an interprocedural, flow-sensitive typestate verification system which is staged to reduce overhead [19]. Bodden *et al.* develop an interprocedural typestate analysis which is flow-sensitive at the intra-procedural level [9]. This is a hybrid system which attempts to eliminate all failure points statically, but uses dynamic checks when necessary. This was later extended to include a backward propagation step that improves precision [8].

Java Bytecode Verification requires a flow-sensitive typing algorithm [33]. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. Like Whiley, the verifier updates the type of a variable after an assignment, and combines types at control-flow join points using a least upper bound operator. However, it does not update the type of a variable after an `instanceof` test. Furthermore, the Java class hierarchy does not form a join semi-lattice. To deal with this, the bytecode verifier uses a simplified least upper bound operator which ignores interfaces altogether, instead relying on runtime checks to catch type errors (see e.g. [32]). However, several works on formalising the bytecode verifier have chosen to resolve this issue with intersection types instead (see e.g. [25,42]).

Gagnon *et al.* present a technique for converting Java Bytecode into an intermediate representation with a single static type for each variable [24]. Key to this is the ability to infer static types for the local variables and stack locations used in the bytecode. Since local variables are untyped in Java bytecode, this is not always possible as they can — and often do — have different types at different points; in such situations, a variable is split as necessary into multiple variables each with a different type.

Dubochet and Odersky [16] describe how structural types are implemented in Scala in some detail, and compare reflexive and generative approaches to implementing methods calls on structural types. They recognise that structural types always impose a penalty on current JVMs, but describe how both techniques generally provide sufficient performance in practice — about seven times slower than Java interface calls in the worse case.

5 Conclusion

The Whiley language implements a flow-sensitive and structural type system on the JVM. This permits variables to be declared implicitly, have multiple types within a function, and be retyped after runtime type tests. The result is a statically-typed language which, for the most part, has the look and feel of a dynamic language. In this paper, we have discussed various details relating to Whiley's implementation on the JVM. Finally, an open source implementation of the Whiley language is freely available from `http://whiley.org`.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proc. OOPSLA*, pages 311–330, 2002.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proc. DLS*, pages 53–64. ACM Press, 2007.

[3] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *Proc. ECOOP*, volume 3586 of *LNCS*, pages 428–452. Springer-Verlag, 2005.

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, pages 57–74. ACM Press, 2006.

[5] F. Barbanera and M. Dezani-CianCaglini. Intersection and union types. In *Proc. of TACS*, volume 526 of *LNCS*, pages 651–674, 1991.

[6] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proc. OOPSLA*, pages 479–498, 2007.

[7] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proc. OOPSLA*, pages 117–136, 2009.

[8] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. ICSE*, pages 5–14, 2010.

[9] E. Bodden, P. Lam, and L. J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proc. ESEC/FSE*, pages 36–47. ACM Press, 2008.

[10] L. Cardelli. Structural subtyping and the notion of power type. In *Proc. POPL*, pages 70–79. ACM Press, 1988.

[11] R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI*, pages 278–292. ACM Press, 1991.

[12] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.

[13] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.

[14] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. POPL*, pages 232–245. ACM Press, 1993.

[15] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. PLDI*, pages 50–62, 2009.

[16] G. Dubochet and M. Odersky. Compiling structural types on the JVM. In *ECOOP 2009 Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICLOOPS)*, 2009.

[17] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

[18] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

[19] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2):1–9, 2008.

[20] C. Flanagan. Hybrid type checking. In *Proc. POPL*, pages 245–256. ACM Press, 2006.

[21] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203. ACM Press, 1999.

[22] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12. ACM Press, 2002.

[23] M. Furr, J.-H. An, J. Foster, and M. Hicks. Static type inference for Ruby. In *Proc. SAC*, pages 1859–1866. ACM Press, 2009.

[24] E. Gagnon, L. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proc. SAS*, pages 199–219, 2000.

[25] A. Goldberg. A specification of java loading and bytecode verification. In *Proc. CCS*, pages 49–58, 1998.

[26] J. Gosling, G. S. B. Joy, and G. Bracha. *The Java Language Specification, 3rd Edition.* Prentice Hall, 2005.

[27] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Proc. OOPSLA*, pages 321–336. ACM Press, 2007.

[28] S. Halloway. *Programming Clojure.* Pragmatic Programmers, 2009.

[29] T. Hoare. Null references: The billion dollar mistake, presentation at qcon, 2009.

[30] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. POPL*, pages 79–90. ACM Press, 2006.

[31] A. Igarashi and H. Nagira. Union types for object-oriented programming. *JOT*, 6(2), 2007.

[32] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[33] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, second edition, 1999.

[34] R. P. Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41(7):22–26, 2008.

[35] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

[36] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. POPL*, pages 228–241, 1999.

[37] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[38] L. D. Paulson. Developers shift to dynamic programming languages. *IEEE Computer*, 40(2):12–15, 2007.

[39] D. J. Pearce and J. Noble. Flow-sensitive types for whiley. Technical Report ECSTR10-23, Victoria University of Wellington, 2010.

[40] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[41] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. CC*, pages 334–554, 2001.

[42] C. Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *Proc. TACAS*, pages 89–103, 1999.

[43] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[44] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. CSF*, pages 186–199, 2010.

[45] The scala programming language. http://lamp.epfl.ch/scala/.

[46] J. Siek and W. Taha. Gradual typing for objects. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 151–175. Springer-Verlag, 2007.

[47] D. Spinellis. Java makes scripting languages irrelevant? *IEEE Software*, 22(3):70–71, 2005.

[48] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[49] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE TSE*, 19(5):478–485, 1993.

[50] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. POPL*, pages 377–388. ACM Press, 2010.