Status Report: Specifying JavaScript with ML

David Herman
Northeastern University
dherman@ccs.neu.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@soe.ucsc.edu

Abstract

The Ecma TC39-TG1 working group is using ML as the specification language for the next generation of JavaScript, the popular programming language for browser-based web applications. This "definitional interpreter" serves many purposes: a high-level and readable specification language, an executable and testable specification, a reference implementation, and an aid in driving the design process. We describe the design and specification of JavaScript and our experience so far using Standard ML for this purpose.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Languages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Documentation, Languages, Standardization

Keywords JavaScript, Standard ML, definitional interpreters

1. Introduction

Thirty-five years ago, John Reynolds wrote of the importance of using *definitional interpreters* as a tool for language specification (Reynolds 1972). Despite the many developments in computer science of mathematical frameworks for formal semantics, this "engineer's approach" to language design and specification still holds relevance today.

In this paper we describe ongoing work in the Ecma TC39-TG1 working group using Standard ML for the specification of the next generation of the ECMAScript programming language, better known as JavaScript (Ecma 1999). JavaScript is a popular programming language for browser-based web applications, made even more popular since the advent of "Ajax" (Garrett 2005), a style of rich, interactive web applications for which JavaScript is the key enabling technology. The fourth edition of the ECMAScript standard represents a major advance in the history of JavaScript, both in the scope of the language design and in the approach to specification, and ML has played an important role in the process.

Our paper proceeds as follows. In Section 2, we describe the essential pieces of the original JavaScript language. In Section 3, we introduce some of the interesting new features of JavaScript 2.0. Section 4 describes the history of approaches to specifying JavaScript, and Section 5 describes the rationale behind using Standard ML as a specification language by contrasting it with other approaches. Section 6 describes some of the specific uses of Standard ML language features for modeling elements of the JavaScript semantics. Section 7 discusses the future of JavaScript and possibilities for further exploration of the language using the Standard ML reference implementation as a tool.

ML'07, October 5, 2007, Freiburg, Germany.

2. JavaScript 1.x

The JavaScript programming language was invented by Brendan Eich at Netscape and first appeared in 1996 in the Netscape Navigator 2.0 web browser. Despite its name, the language has little to do with the Java programming language beyond minor syntactic similarities. JavaScript is a dynamically typed, prototype-based object-oriented programming language with mostly lexical scope and first-class function closures.

The language was first standardized at Ecma International in 1997 under the name ECMAScript. The third edition of the ECMAScript specification, published in 1999, is the current standard and forms the basis of all major implementations of JavaScript, including those of Mozilla Firefox, Microsoft Internet Explorer, the Opera web browser, Apple Safari and WebKit, Rhino (now shipping as an extension language with the Java standard library), and the Adobe/Macromedia Flash scripting language ActionScript.

In the remainder of this section we give a brief introduction to the JavaScript programming language. Examples shown at an interactive shell are prefixed with a prompt ("`> `").

Objects

The primary datatype of JavaScript is the object, essentially an associative array or table of *properties* with fast lookup and update:

```
> var o = { name: "Alice", age: 41 };
> ++o.age
42
```

Functions

Functions are first-class values in JavaScript (and in fact are themselves objects with property tables). Function objects close over their lexical environment.

Object methods are nothing more than properties whose values happen to be functions. This results in a somewhat subtle interpretation of references to `this`. The interpretation of a method call

$expr\,.\,ident(val, \ldots)$

is to look up the function bound to $expr.ident$, and invoke it with `this` bound to the result of the left-hand side expression $expr$.

```
> var o = { name: "Alice",
            age: 42,
            toString: function() {
                        return ("[employee "
                                + this.name
                                + "]");
                      } };
> print(o.toString());
[employee Alice]
```

Because methods are unrestricted function values and may be bound to ordinary lexical variables, there are no restrictions on the appearance of `this`. Rather, in ordinary function calls, the binding of `this` defaults to the single "global object," an ambient object whose properties include the top-level variable bindings.

```
> var name = "Nobody";
> var f = o.toString;
> print(f());
[employee Nobody]
```

Prototypes

The object system of JavaScript is based on prototypes, inspired by the Self programming language (Ungar and Smith 1987). At runtime, every object has an implicit link to another object, its *prototype*. Property lookup (for dereference, assignment, or method call) recursively searches the "prototype chain," providing a dynamic form of inheritance.

Programmers can explicitly construct the prototype relationships of new objects by creating *constructor functions*. Any function in JavaScript can serve as a constructor. The runtime system uses the `prototype` property of the function object to create the internal prototype link of new objects. As a simple example, consider a definition of an `Employee` constructor:

```
function Employee(name, age) {
    this.name = name;
    this.age = age;
}
Employee.prototype = {
    toString: function() {
        return ("[employee "
                + this.name
                + "]");
    }
};
```

Every instance of `Employee` inherits the properties of the prototype, namely the `toString` method:

```
> var a = new Employee("Alice", 42);
> var b = new Employee("Bob", 34);
> print(a.toString());
[employee Alice]
> print(b.toString());
[employee Bob]
```

Notice that `this` is bound to the newly created object in the body of the constructor function when called with `new`.

3. JavaScript 2.0

The work in progress on ECMAScript Edition 4/JavaScript 2.0 represents a significant revision to the language (Horwat 2001). The advent of Ajax (Garrett 2005) has increased the popularity and prevalence of increasingly complex web applications. As JavaScript applications have grown in maturity and sophistication, so too must the language to keep up with the needs of its community.

Class-based OOP

One of the most common idioms found in contemporary JavaScript applications and frameworks is emulation of class-based object-orientation through the prototype system. JavaScript 2.0 is therefore standardizing a system of classes and interfaces similar to Java or C♯ (Gosling et al. 2000; Ecma 2006).

Name management

The only reliable form of information hiding in JavaScript 1.x is lexical scope. Because all object properties are accessed by string names, it is not possible to prevent clients from guessing or discovering object properties that are meant to be internal.

To provide some name control while retaining backwards-compatibility, JavaScript 2.0 is generalizing property names from simple strings to pairs of special *namespace* values and string names. When the namespace is left unspecified, a default namespace is used to preserve backwards-compatible behavior. However, programs can generate lexically scoped, hidden namespaces and explicitly store private data in properties keyed by these namespaces.

Gradual typing

With the desire to create sophisticated applications comes the need to document and check program invariants as types. JavaScript 2.0 is introducing a static type system with both nominal types to support the class-based portion of the language and structural types such as function types, array types, and record-like "object types," to support functional and lightweight object-oriented programming idioms.

Naturally, backwards-compatibility demands that dynamically-typed programs continue to work unchanged. Some of the legacy constructs of the language, such as the prototype system, are notoriously difficult to fit into a static typing discipline (Anderson et al. 2005; Thiemann 2005). Furthermore, common idioms in web applications involve dynamic constructs such as loading and evaluation. For all these reasons, JavaScript 2.0 needs to support flexible interoperation between static and dynamic typing. This style of type system has become popularly known as *gradual typing* (Siek and Taha 2006, 2007; Herman et al. 2007). As of July 2007, there is still ongoing discussion regarding what notion of gradual typing will be supported by JavaScript 2.0.

Control constructs

JavaScript 1.x already contains several non-local control constructs, including `for`, `while`, and `do` loops, the standard `break` and `continue` loop control operators, simple untyped exceptions, and `return`. JavaScript 2.0 is introducing several new control constructs that we model in Standard ML.

In JavaScript 2.0, function calls that are not within the scope of an exception handler and that appear in tail position with respect to the `return` operator are specified to be tail calls. There are several subtleties involved in designing tail calls for JavaScript, including function return-type checks in gradually-typed languages (Herman et al. 2007).

JavaScript 2.0 is introducing a form of coroutines called *generators*, based on a construct from Python (Schemenauer et al. 2001; van Rossum and Eby 2005). A generator is a function that uses the new `yield` keyword to suspend its current activation and return an intermediate value to the calling context. Callers can resume the suspending activation, passing in a value for the `yield` expression to evaluate to. We discuss the specification of generators in Section 6.

4. Evolution of ECMAScript specifications

Previous editions of the ECMAScript standard used a simple, imperative pseudocode as a meta-language for semantics specification, such as the example shown in Figure 1. This meta-language was never formally specified, but its semantics was perhaps simple enough to be inferred. Nevertheless, the low level of abstraction and imperative nature of the language (including heavy use of mutation and "go to") sometimes resulted in rather obfuscated pseudocode.

Grammar production:
$ConditionalExpression \rightarrow LogicalORExpression$ `?` $AssignmentExpression$ `:` $AssignmentExpression$

Evaluation:

1. Evaluate $LogicalORExpression$.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, go to step 8.
5. Evaluate the first $AssignmentExpression$.
6. Call GetValue(Result(5)).
7. Return Result(6).
8. Evaluate the second $AssignmentExpression$.
9. Call GetValue(Result(8)).
10. Return Result(9).

```
fun evalCondExpr (regs:REGS)
                 (cond:EXPR)
                 (thn:EXPR)
                 (els:EXPR)
    : VAL =
    let
        val v = evalExpr regs cond
        val b = toBoolean v
    in
        if b
        then evalExpr regs thn
        else evalExpr regs els
    end
```

Figure 1. Pseudocode from ECMAScript Edition 3 (left) compared with corresponding Standard ML code (right).

Even more problematic was the fact that the pseudocode was not executable, which precluded testing. This resulted in quite a few bugs in the standard (Horwat 2003a).

Due to the limitations of earlier, informal specification mechanisms, there was a clear desire on the part of the committee for some sort of formal or executable specification, whereby the ability to execute this specification on a variety of JavaScript programs would help detect errors early in the language design process, and would provide additional confidence in the correctness, completeness, and consistency of the final specification.

In the initial stages of development of ECMAScript Edition 4, Waldemar Horwat addressed the lack of precision in previous standards by defining an Algol-like, typed metalanguage. Early proposals used this metalanguage to specify the language constructs, and an implementation in Common Lisp served as an early reference implementation (Horwat 2003b,c). Horwat attempted to provide a denotational interpretation for the types and terms of the metalanguage, but this proved unwieldy.

Beginning in early 2006, we explored the use of term-rewriting languages such as Stratego (Visser 2001) or PLT Redex (Matthews et al. 2004) to develop an executable operational semantics. In order to accomodate the non-trivial static semantics and syntactic sugar in the language, we considered designing yet another intermediate language that would be close in flavor to the pseudocode in previous specifications, while still being fully formalized. However, this approach would essentially have required designing *two* languages concurrently (ECMAScript Edition 4 and its specification language), introducing significant additional work and perhaps unnecessary complexity.

Hence, in November 2006, we decided to use an existing programming language as the specification language for Edition 4, with ML being an obvious choice for the specification language. There was some subsequent discussion of which dialect of ML to use, with the committee initially leaning towards OCaml (in part due to somewhat better tool support, error messages, etc), but eventually choosing SML (based in part on arguments that it is a more mature language and is formally specified (LtU 2006)).

Over the next several months, much of the work of the committee became essentially a software engineering effort, based around a version control system (Monotone 2007) and, later, a bug tracking database (Trac 2007). This work largely has involved reifying the current language design as code. There has been a fair amount of discussion of various implementation details (for example, the exact structure of abstract syntax tree), and a rather surprising amount of iteration and refactoring. The reference implementation is now over 20 KLOC of ML code, with the main phases being:

- parsing (7 KLOC),
- a definition phase that includes name resolution and identifying compile-time constants (3 KLOC),
- type checking (2 KLOC),
- and evaluation (5 KLOC).

In addition, there is around 10KLOC of ES4 code that defines most of the Javascript standard libraries. Writing this code in ES4 helps reduce the complexity of the core semantics, with some cost in performance.

A pre-release of this reference implementation is available at `http://www.ecmascript.org/download.php`.

5. Language specification styles

In this section we briefly reflect on our experiences to date in using ML to write a definitional interpreter for ECMAScript Edition 4, and compare this approach with two commonly-used alternatives:

- informal prose, such as is used in the Java Language Specification (Gosling et al. 2000);
- formal, mathematical specifications, such as that used to specify Standard ML (Milner et al. 1997).

Thus, the choice of language specification styles can be succinctly summarized as

Code vs. Prose vs. Math

5.1 Language specifications: Code vs. Prose

Our initial discussions before November 2006 almost exclusively used prose, together with some JavaScript code fragments, both in person, on whiteboards, and on a wiki (Ecma 2007). Many of these discussions were at a fairly high level, and assumed a fairly substantial amount of background knowledge regarding JavaScript implementations. As might be expected, underlying assumptions were often left implicit and occasionally mis-understood, and the interactions between various features were not always explored in complete detail.

This communication style worked well early in the design process, as various design alternatives were being compared, and there

was little benefit to fully formalizing a design alternative that may later be discarded.

Once we switched to a definitional interpreter, the interaction style of the committee changed substantially, from monthly $1\frac{1}{2}$-day discussion-oriented meetings to 3-day "hackathons," interspersed with technical discussions, as various corner cases in the language design and implementation were discovered and resolved. The definitional interpreter worked well in forcing the committee to clarify many unspoken assumptions, and provided a concrete artifact that grounded many discussions that might otherwise have been overly abstract. It also provided valuable implementation experience for Edition 4.

The style of code is an important aspect of a definitional interpreter. Overall, there was fairly clear agreement on "clarity over performance," that is, the primary goal of the definitional interpreter is to be define the language specification, rather than describe a realistic, efficient implementation of that language. We strive to emphasize clarity, readability, and abstractness in our code, never efficiency. Of course, this results in a slow implementation, but the purpose of the reference implementation is specification rather than usability.

Another important guideline we have followed is to keep the core semantics as small as possible by modeling most of the standard library in JavaScript, minimizing the reliance on "magic" hooks into the semantics. For example, the reference implementation does not implement regular expressions natively in ML, even though most realistic implementations would do so to improve performance.

As might be expected, writing a definitional interpreter for a large and realistic language such as ECMAScript Edition 4 involved a substantial time investment, and required significant communication and co-operation by committee members. This time investment included both essential and accidental complexity (Brooks 1986): the essential complexity being the actual cost of specifying the language semantics in full detail; the accidental complexity included the learning curve with SML and its tool suite, wrestling with unintuitive parts of the SML language, and dealing with imprecise error messages (eg, "there is a type error somewhere in these 200 lines of code"). We partially overcame the latter problem by providing explicit types for all top-level functions. Also, the SML module system provides limited support for mutually-recursive modules, with the result that mutually-recursive "conceptual" modules must be sometimes coalesced into a monolithic SML module, with some loss in clarity.

Overall, despite the overheads and costs of the definitional interpreter, our experience to date suggests that it works much better in several regards (consistency, completeness, implementation experience, early defect detection, etc) than an informal English specification.

5.2 Language specifications: Code vs Math

The definitional interpreter has essentially two goals:

- to precisely *define* the language semantics, and
- to *communicate* this semantics to the intended audience (to other committee members, to language implementors, and to other language users).

Other language definition styles, such as operational or denotational semantics, could also have satisfied the first goal but not the second, in large part because mathematical semantics involves specialized notation that is unfamiliar to large parts of the target audience. (Additional formal notations would be necessary to also specify the type system of the language.)

This limitation became quite clear in the committee's discussions of lightweight strategies for gradual typing. Our English discussions and descriptions always felt overly vague and imprecise. One of the authors (Flanagan) developed several formal models of the operational semantics and type systems for gradual typing, but these were inaccessible to many committee members. More recently, we developed a definitional interpreter for the gradually-typed lambda calculus, which finally provided a concise and precise description that all committee members could understand and discuss. That is, in this instance, code succeeded where prose and math had both failed.

Many of the committee members found formal semantics daunting, especially working on an aggressive timeline. However, every single member of the committee is an expert programmer. Expressing semantics in a programming language, albeit unfamiliar to some, turned out to be far more accessible than many semantics formalisms. This allowed more committee members to contribute to the specification rather than leaving a small subset of the members on the critical path. We expect this will have benefits for the readability of the specification as well, since more people in the target audience are likely to be familiar with functional programming than with formal semantics.

Definitional interpreters do work at a somewhat lower level of abstraction than operational or denotational semantics, in part because they deal with more low-level details. Nevertheless, we believe that it has been significantly easier for the committee to formalize the Edition 4 semantics as code than as mathematics, because:

1. it requires much less specialized training;
2. it leverages prior experience on programming language implementation (as opposed to semantics);
3. SML provides various linguistic features (side effects, callcc, etc) that have proven quite useful; and
4. as mentioned above, type systems and test suites are invaluable in debugging the language semantics.

5.3 Language specifications: Code *and* Prose

The increased precision of code over prose can also be a drawback: because code operates at a lower level of abstraction than semantics, it can result in overspecification. For example, libraries often leave portions unspecified to allow for multiple implementation strategies; but an actual implementation does not have the freedom to leave anything undefined. Often such implementation decisions are observable to user programs. For instance, a library function may document its result type as an abstract interface, but reflection facilities would allow programs to observe the concrete class used to implement that interface.

To avoid overspecification, the reference implementation does not stand on its own as a complete specification, and parts of it will not even be included in the normative standard. Rather, the document will excerpt portions of the interpreter where appropriate, surrounding code with prose where necessary. The reference implementation will likely be provided as an informative appendix or companion document.

6. Implementation overview

In this section we describe some of the techniques we use for modeling JavaScript features in ML. Because of the feature set of Standard ML, it is possible for us to model JavaScript in a direct style, using the implicit control and store of ML to model those of JavaScript. Of course, we could write the interpreter in continuation-passing and store-passing style, using ML as little more than an executable lambda calculus. This would bring the model closer to a formal semantics. Indeed, in some cases the price we pay for direct style is the need for somewhat less natural models

```
datatype VAL = Object of OBJ
             | Null
             | Undef

     and OBJ =
         Obj of { ident: OBJ_IDENT,
                  tag: VAL_TAG,
                  props: PROP_BINDINGS,
                  proto: VAL ref,
                  magic: MAGIC option ref }

     and VAL_TAG =
         ObjectTag of FIELD_TYPE list
       | ArrayTag of TYPE_EXPR list
       | FunctionTag of FUNC_TYPE
       | ClassTag of NAME

     and MAGIC =
         UInt of Word32.word
       | Int of Int32.int
       | ...

withtype OBJ_IDENT = int

     and PROP = { ty: TYPE_EXPR,
                  attrs: ATTRS,
                  state: VAL }

     and PROP_BINDINGS = (NAME * PROP) list ref
```

Figure 2. Definition of runtime values in ECMAScript Edition 4.

of individual features. But writing in direct style allows us to keep these representations localized, resulting in a more modular and comprehensible language definition.

6.1 Features of ML

In contrast with past specifications, the vast majority of the ECMAScript Edition 4 reference implementation is written in a pure functional style. This greatly improves the clarity and raises the level of abstraction of the specification (see Figure 1). Nevertheless, a moderate use of side effects can be appropriate for modeling language features. As it turns out, the side effects of ML are well-suited to model the primary effects of JavaScript.

Reference cells

Reference cells provide a natural model of JavaScript's mutable variables and object properties. Figure 2 shows most of the data definition for ECMAScript Edition 4 values in Standard ML. There are two special sentinel values in ECMAScript Edition 4, `null` and `undefined`; all other values are of the `Object` variant of `VAL`. Every object has a unique identity, a runtime type tag, a mutable table of properties, a link to a prototype object, and an optional "magic" internal datum. This latter field is used to handle values of primitive types such as integers and floating point numbers.

Exceptions

JavaScript includes a number of non-local jumps that can be modeled as ML exceptions, from JavaScript exceptions to loop breaks and function return. Modeling tail calls is somewhat more subtle, since installing an exception handler at the entry of every procedure activation precludes using a simple tail call to an ML function. Instead, we model a tail call as a modified trampoline (Ganz et al. 1999): the function performing a tail call raises an exception of type

```
exception TailCallException of (unit -> VAL)
```

with a thunk to invoke the tail function. The semantics of non-tail function calls includes wrapping the call in a handler that catches instances of `TailCallException` and invokes the associated thunk.

First-Class Continuations

The one non-standard feature of SML that we are considering exploiting is `callcc`. Because the semantics of generators (see Section 3) involves suspending and reifying a delimited portion of the current continuation, some amount of reification of control is necessary. To convert the entire interpreter to continuation-passing style just to support this one, largely orthogonal language feature would be unfortunate. Instead, we could use a non-native encoding of the delimited continuation operators `shift` and `reset` implemented with native `callcc` (Herman 2007). While non-standard, the semantics of continuations are well understood and widely implemented.

6.2 Engineering the reference implementation

The current pre-release of the reference implementation is built with Standard ML of New Jersey (Appel and MacQueen 1991). We are currently working on ports to MLton (Weeks 2006) and SML.NET (Benton et al. 2004). Porting to multiple implementations of SML has helped us to discover non-standard features we used unwittingly, improving portability and forcing us to code to the standard language. We also hope to reap the benefits each implementation has to offer, specifically performance from MLton and interoperability from SML.NET.

The reference implementation is already delivering on its promise to help with testing. The first and probably most important tests we have performed are regression tests: both Mozilla and Adobe have contributed sizeable test suites from their own implementations of ECMAScript. The current build passes more than 93% of the Mozilla regression tests. The failing test cases are caused both by out-of-date tests (written for previous versions of the language), or tests that use features that the reference implementation does not yet implement correctly.

7. What's next

Since SML is a formally-specified language (Milner et al. 1997), we argue that the definitional interpreter provides a fully-formal semantics, if sometimes overspecified. We hope this definitional interpreter will provide a foundation for further research. In particular, it enables us to concisely formalize type soundness for JavaScript:

> *if a JavaScript program passes the type checker,*
> *then certain run-time errors do not occur.*

Given the size of the definitional interpreter (upwards of 20 KLOC), unsurprisingly type soundness remains unproven. However, various forms of model checking could be applied to discover counter-examples to type soundness (Darga and Boyapati 2006), or to provide additional confidence in the implementation. Eventually, we hope interactive (but highly automated) theorem provers could be used to formally verify type soundness for our language specification.

Acknowledgements: We would like to thank Lars Hansen, Brendan Eich, and the anonymous reviewers for valuable feedback on this paper. This work was partially supported by a Sloan Fellowship and a grant from the Mozilla Corporation.

References

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pages 428–453, 2005.

Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer Verlag, 1991.

Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the SML.NET experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 215–226, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-819-9.

Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. In *Information Processing 86*, pages 1069–1076, 1986. International Federation of Information Processing (IFIP) Congress '86.

Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 363–382, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4.

Ecma 2007. ECMAScript Edition 4 specification wiki, 2007. URL `http://wiki.ecmascript.org`.

Ecma 2006. *C♯ Language Specification*. Ecma International, 4th edition, 2006. ECMA-334.

Ecma 1999. *ECMAScript Language Specification*. Ecma International, 3rd edition, 1999. ECMA-262.

Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

Jesse J. Garrett. Ajax: A new approach to web applications, 2005.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, Boston, Mass., 2000. ISBN 0-201-31008-2.

David Herman. Functional Pearl: The Great Escape. In *International Conference on Functional Programming (ICFP)*, October 2007. To appear.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming*, April 2007.

Waldemar Horwat. ECMAScript edition 3 errata, June 2003a. URL `http://www.mozilla.org/js/language/E262-3-errata.html`.

Waldemar Horwat. JavaScript 2.0: Evolving a language for evolving systems. URL `http://www.mozilla.org/js/language/evolvingJS.pdf`. Lightweight Languages Workshop (LL1), 2001.

Waldemar Horwat. ECMAScript 4 Netscape proposal, June 2003b. URL `http://www.mozilla.org/js/language/old-es4`.

Waldemar Horwat. JavaScript 2.0 experimental semantics, 2003c. URL `http://lxr.mozilla.org/mozilla/source/js2/semantics/`.

LtU 2006. Specifying ECMAScript via ML, November 2006. URL `http://lambda-the-ultimate.org/node/1784`.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications (RTA2004)*, 2004.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997. ISBN 0262631814.

Monotone 2007. Monotone: Distributed version control, 2007. URL `http://monotone.ca/`.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, New York, NY, August 1972. ACM Press.

Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Simple generators, May 2001. URL `http://www.python.org/dev/peps/pep-0255/`. PEP-255.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, Berlin, Germany, July 2007.

Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium On Programming*, pages 408–422, 2005.

Trac 2007. The Trac Project, 2007. URL `http://trac.edgewall.org/`.

David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, New York, NY, 1987. ACM Press.

Guido van Rossum and Phillip J. Eby. Coroutines via enhanced generators, May 2005. URL `http://www.python.org/dev/peps/pep-0342/`.

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

Stephen Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 Workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-483-9.