ISSN 2186-7437

NII Shonan Meeting Report

No. 2014-6

Software Contracts for Communication, Monitoring, and Security

Atsushi Igarashi (Kyoto University, Japan)
Peter Thiemann (University of Freiburg, Germany)
Philip Wadler (University of Edinburgh, UK)

May 26–30, 2014

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Software Contracts for Communication, Monitoring, and Security

Atsushi Igarashi (Kyoto University, Japan)
Peter Thiemann (University of Freiburg, Germany)
Philip Wadler (University of Edinburgh, United Kingdom)

May 26–30, 2014

The contract metaphor is the basis of today ' s software verification. Expressing invariants, pre-, and postconditions with logical formulae as part of an interface has become common fare in the construction of dependable software. Often, general logical formulae are replaced by type qualifiers or other abstractions to soften the learning curve.

While traditional contracts focus on characterizing behavior in terms of snapshots by relating the prestate of a function to its poststate, a wealth of recent work has considered behavioral contracts like session types that describe the temporal behavior of program components and/or generalize the notion of the observed state. Often this kind of contract describes the communication behavior of a group of components, but there are many other use cases like typestate reasoning or the description of security policies.

We observed that there are different communities (contracts, sessions, security and typing, specification, and static vs. dynamic verification) that use similar approaches to pursue similar research objectives. Hence, the goal of the seminar is to explore the potential for cross fertilization, to identify commonalities and differences, to get inspiration by discussing the techniques used in the other fields, and to find grounds for future collaboration. The next few paragraphs substantiate this claim by giving very brief outlines of ongoing work in these communities.

Session Types Session calculi have been discovered by Kohei Honda in his 1993 paper " Types for Dyadic Interaction " . They extend process calculi with session types that provide a statically checkable (or occasionally dynamically checked), structured means of specifying synchronous communication between two parties. A session type defines the order of the communication actions on a given communication channel and it prescribes the types of the transmitted values.

A session type may specify labeled alternative behaviors and loops, similar to a regular language.

Meanwhile, numerous variations of session calculi have been developed. Some are based on functional and object-oriented calculi. Others include subtyping, asynchronous communication, or they govern multi-party communications. Recent work has established connections with linear logic as a semantic basis for session types and communication automata, which are widely used in the analysis of network protocols.

Typestate A closely related line of work considers behavioral types and typestate, where the type of an object changes over time to indicate its current internal state. Methods of the object may be enabled or disabled depending on its current type. The prototypical example is the file that must first be opened, then may be read and written repeatedly, and must finally be closed.

Monitoring On the other hand, researchers in computer security have considered application monitoring for a long time. A monitor runs alongside the application program and changes state triggered by a stream of events from the running program. If the monitor is used for auditing, then the event stream is condensed to a log and saved for post-mortem analysis. However, the monitor may also be used to enforce a security policy by terminating the program as soon as it attempts to perform an illegal sequence of events.

Most of the time monitoring is performed at run time, but there is also scope for static monitoring. A prime example in this direction is Walker ' s " A type system for expressive security policies " , which applies linear and dependent types to express finite state monitoring on the type level.

Recent work indicates that session types and typestate are converging (and some of the convergence is organized in the EU-COST action BETTY), but the situation is less clear for the interaction between session typing and security monitoring. In particular, session typing is mostly considered in a static typing context whereas security monitoring usually happens at run time. There are only a few exceptions to each, such as Yoshida ' s work on run-time enforcement of session types and Walker ' s work on a static type system for monitoring. There may even be room for hybrid approaches that apply a mix of static and dynamic checks, in close analogy to existing work in dynamic typing.

1 List of participants

- Massimo Bartoletti, University of Cagliari, Italy
- Ilaria Castellani, INRIA, France
- Stephen Chong, Harvard University, USA
- Christos Dimoulas, Harvard University, USA
- Tim Disney, University of California, Santa Cruz, USA
- Robby Findler, Northwestern University, USA
- Cormac Flanagan, University of California, Santa Cruz, USA
- Ronald Garcia, University of British Columbia, Canada
- Simon Gay, University of Glasgow, UK
- Michael Greenberg, Princeton University, USA

- Reiner Hähnle, TU Darmstadt, Germany
- David Van Horn, University of Maryland, USA
- Atsushi Igarashi, Kyoto University, Japan
- Naoki Kobayashi, University of Tokyo, Japan
- Jay McCarthy, Brigham Young University, USA
- Luca Padovani, University of Torino, Italy
- Jeremy Siek, Indiana University, USA
- Martin Sulzmann, Karlsruhe University of Applied Sciences, Germany
- Asumu Takikawa, Northeastern University, USA
- Peter Thiemann, University of Freiburg, Germany
- Sam Tobin-Hochstadt, Indiana University, USA
- Vasco T. Vasconcelos, University of Libon, Portugal
- Philip Wadler, Edinburgh University, UK
- David Walker, Princeton University, USA
- Nobuko Yoshida, Imperial College London, UK

2 Program

May 25 (Sun.)	
19:00–21:00	Welcome Reception
May 26 (Mon.)	
9:00–10:30	Two-minute introductions
	(break)
11:00–12:00	Talk by Vasconcelos
	(lunch)
13:30–15:30	Talks by Vasconcelos (contd.), Findler, and McCarthy
	(break)
16:00–18:00	Talks by Yoshida and Wadler
May 27 (Tue.)	
9:00–10:30	Talks by Kobayashi and Walker
	(break)
11:00–12:00	Talk by Dimoulas
	(lunch)
13:30–15:30	Talks by Disney, Gay, and Sulzmann
	(break)
16:00–18:00	Talks by Chong, Tobin-Hochstadt, and Takikawa
May 28 (Wed.)	
9:00–10:30	Talks by Siek and Greenberg
	(break)
11:00–12:00	Talks by Hähnle, and Garcia
	(lunch, excursion, and dinner)

May 29 (Thu.)	
9:00–10:30	Talks by Padovani and Bartoletti
	(break)
11:00–12:00	Talks by Thiemann ad Vasconcelos
	(lunch)
13:30–14:30	Talks by McCarthy and Wadler
14:30–15:30	Discussion on noninterference
	(break)
16:00–17:00	Talks by Igarashi and Van Horn
17:00–18:00	Discussion on session types vs contracts
May 30 (Fri.)	
9:00–10:30	Talks by Bartoletti and Castellani
	(break)
11:00–12:00	Wrap-up discussion

3 Overview of Talks

Space-efficient Manifest Contracts

Michael Greenberg, Princeton University, USA

The standard algorithm for higher-order contract checking can lead to unbounded space consumption and can destroy tail recursion. In this work, we show how to achieve space efficiency for contract checking. Working in a manifest context, we define a family of languages: classic λ_H, which is inefficient; forgetful λ_H, which is efficient but skips some checks; and heedful λ_H, which is efficient but may change blame labels. We show first that if classic λ_H produces a value, then so does forgetful λ_H (but not vice versa); we then show that classic and heedful λ_H yield identical values, but possibly differing blame labels.

Fully Abstract Method Contracts

Reiner Hähnle, TU Darmstadt, Germany

A major obstacle facing adoption of formal software verification is the difficulty to track changes in the target code and to accomodate them in specifications and in verification arguments. We introduce fully abstract method contracts, a new verification rule for method calls that can be used in most contract-based verification settings. The rule makes it possible to reason about programs without having to know the implementation of called methods. By combining abstract method contracts and caching of verification conditions, it is possible to detect reusability of contracts automatically via first-order reasoning. This is the basis for a verification framework able to deal with code undergoing frequent changes. Fully abstract contracts have been implemented in the Java verification tool KeY. We report on experiments that show their saving potential.

Blame Concisely

Philip Wadler, Edinburgh University, UK

An introduction to the blame calculus and summary of contributions, covering the following.

> Well-typed programs can't be blamed, Wadler and Findler, ESOP 2009.
>
> Threesomes, with and without blame, Siek and Wadler, POPL 2010.
>
> Blame for all, Ahmed, Findler, Siek, and Wadler, POPL 2011.
>
> Blame, threesomes, and coercions, precisely, Siek, Thiemann, and Wadler, submitted.
>
> Symmetric blame, Wadler, in preparation.

Session Types and Typestate for Object-Oriented Programming

Simon Gay, University of Glasgow, UK

Session types describe the sequence and format of messages on point-to-point communication channels. They allow communication protocols to be specified type-theoretically so that their implementations can be verified by static type-checking. Session types were introduced in the setting of process calculus, and later formulated for imperative, functional and object-oriented languages. The quest for a clean integration of session types and objects leads to the idea of embedding session types in a more general setting of typestate, in which the operations available for a given object are state-dependent. We are exploring this idea with a typestate system in which typestate specifications are inspired by session types, and developing an experimental implementation as an extension of Java.

Joint work with Vasco Vasconcelos, Antonio Ravara, Nils Gesbert, Ornela Dardha, Dimitrios Kouzapas

Secure shell scripting

Steven Chong, Harvard University, USA

Reasoning about the security of shell scripts is notoriously hard. This is in large part because it is difficult for programmers to deduce the effects of shell scripts on the underlying operating system. First, resource references, such as file paths, are typically resolved lazily and subject to race conditions. Second, shell scripts are typically run with the same privileges as the invoking user, making it hard to determine or enforce that a script has all (and only) permissions to execute successfully. Third, shell scripts invoke other programs, often arbitrary binaries.

In this talk, I present the preliminary design and implementation of Shill, a secure shell scripting language that uses fine-grained capabilities to restrict access to resources. Capabilities bind resources at the time of their creation, and avoid vulnerabilities arising from lazy name resolution. Shill scripts come with contracts that specify and restrict what capabilities the script may use. A Shill script can invoke an arbitrary binary in a sandbox that limits the privileges of the binary based on a set of capabilities. Capabilities together with contracts and sandboxing enable the caller of a script to reason precisely about which resources a script (and the binaries it calls) may access, and thus, Shill helps reason safely and effectively about the use and composition of scripts. We have implemented Shill on top of FreeBSD, using Racket and the FreeBSD Trusted MAC framework.

Complete contract monitors and their applications

Christos Dimoulas, Harvard University, USA

Contracts are a popular mechanism for enhancing the interface of components. In the world of first-order functions, programmers embrace contracts because they write them in a familiar language and easily understand them as a pair of a pre-condition and a post-condition. In a higher-order world, contracts offer the same expressiveness to programmers but their meaning subtly differs from the familiar first-order notion. For instance, it is unclear what the behavior of dependent contracts for higher-order functions or of contracts for mutable data should be. As a consequence, it is difficult to design monitoring systems for such higher-order worlds.

In response to this problem, we investigate complete monitors, a formal framework for deciding if a contract system is correct. The intuition behind the framework is that a correct contract system should:

- mediate the exchange of values between contracted components
- and blame correctly in case of contract violations.

The framework reveals flaws in the semantics for dependent contracts from the literature and suggests a natural fix. In addition, we demonstrate the usefulness of the framework for language design with a language with contracts for mutable data and a language that mixes typed and untyped imperative programs. The final contribution is the provably correct design of a novel form of contracts, dubbed options contracts, that mix contract checking with random and probabilistic checking.

Multiparty session types and their applications to a large distributed system

Nokuko Yoshida, Imperial College London, UK

We give a summary of our recent research developments on multiparty session types for verifying distributed and concurrent programs, and our collaborations with industry partners and a major, long-term, NSF-funded project (Ocean Observatories Initiatives) to provide an ultra large-scale cyberinfrustracture (OOI CI) for 25-30 years of sustained ocean measurements to study climate variability, ocean circulation and ecosystem dynamics.

We shall first talk how Robin Milner, Kohei Honda and Yoshida started collaborations with industry to develop a web service protocol description language called Scribble and discovered the theory of multiparty session types through the collaborations. We then talk about the recent developments in Scribble and the runtime session monitoring framework currently used in the OOI CI.

Deductive Verification of Communication Contracts

Vasco T. Vasconcelos, University of Libon, Portugal

Communication contracts describe the global interactive behaviour of concurrent and distributed systems. The challenge is to verify whether a given program (or collection of programs) adheres a contract. If that turns out to be the case, properties such as absence of 'message not understood', undesired races and deadlocks are automatically guaranteed. We show one such system for Message Passing Interface programs.

Secure information flow for synchronous reactive programs

Ilaria Castellani, INRIA, France

We consider the security property of noninterference in a core synchronous reactive language that we call CRL. In the synchronous reactive paradigm, threads communicate by means of broadcast events, and their parallel execution is regulated by a notion of instant. Threads get suspended while waiting for an absent event or when deliberately releasing the control for the current instant. An instant is a period of time during which all threads compute up to termination or suspension. A program interacts with its environment only at the start and the end of instants.

We define two bisimulation equivalences on CRL programs, corresponding respectively to a fine-grained and to a coarse-grained observation of programs. Based on these bisimulations, two properties of Reactive Noninterference (RNI) are introduced, formalising secure information flow. Both RNI properties are clock-sensitive (events are observed with their clock-stamp, i.e. the instant in which they are produced) and termination-insensitive. Coarse-grained RNI is more abstract than fine-grained RNI in that it ignores the order of generation of events and repeated emissions of the same event within an instant.

We present a type system guaranteeing both security properties. Thanks to some design choices of CRL and to specific typing rules for conditionals, this type system allows for a precise treatment of termination leaks, which are an issue in parallel languages.

Finally, we try to draw some analogies between the behaviour of synchronous programs within instants and that of pi-calculus processes within sessions.

Constructive Trace Validation with Regular Program Properties

Martin Sulzmann, Karlsruhe University of Applied Sciences, Germany

We consider the problem of verifying if a program trace is valid with respect to a regular program specification, e.g. expressed in terms of regular expressions or linear temporal logic formulas. Existing methods only provide yes/no answers. Such answers are unsatisfactory and not helpful to precisely track down the source of a failure. We introduce a constructive method which provides detailed explanations about the outcome of trace validation in a form comprehensible for the user. For example, we obtain information which parts of the specification have been exercised in case of success and which parts have been violated in case of failure. The approach has been fully implemented and is applied in some real-world projects for testing of synchronously executed software components.

Types and Effects for Deadlock-Free Higher-Order Concurrent Programs

Luca Padovani, University of Torino, Italy

Deadlock freedom is for concurrent programs what progress is for sequential ones: it indicates the absence of stable (i.e., irreducible) states in which some pending operations cannot be completed. In the particular case of communicating processes, operations are inputs and outputs on channels and deadlocks may be caused by mutual dependencies between communications. In this talk we see how to define an effect system ensuring deadlock freedom of higher-order programs and discuss some of its properties.

From Behavioral Types to Higher-Order Model Checking

Naoki Kobayashi, University of Tokyo, Japan

The talk covers three topics related (in my opinion) to the seminar: behavioral types, resource usage verification (or typestate checking), and higher-order model checking, and discuss their relationship. First, I review our earlier work on behavioral types, and explain how they can be used for analysis of concurrent programs and resource usage verification (or, typestate checking) of functional programs. I will then explain how that line of work has evolved to our more recent work on resource usage verification based on higher-order model checking.

Design and Evaluation of Gradual Typing for Python

Jeremy Siek, Indiana University, USA

I present Reticulated Python, a lightweight system for experimenting with gradual-typed dialects of Python. The dialects are syntactically identical to Python 3 but give static and dynamic semantics to the type annotations already present in Python 3. Reticulated Python consists of a typechecker, a source-to-source translator that inserts casts, and Python libraries that implement the casts. Using Reticulated Python, we study a gradual type system that features structurally-typed objects and type inference for local variables. We evaluate two dynamic semantics for casts: one based on Siek and Taha (2007) and a new design, using transient casts, that does not require proxy objects. We evaluate these designs with two third-party Python programs: the CherryPy web framework and a library of statistical functions.

Manifest Contracts for Algebraic Datatypes

Atsushi Igarashi, Kyoto University, Japan

Manifest contract calculi are a family of statically typed higher-order contract calculi where contract information occurs as refinement types—e.g., $\{x : \mathit{Int} \mid 0 < x\}$ means positive integers. Contracts in such calculi are checked statically if possible; otherwise dynamically with type conversion, also called casts. It is a natural idea to apply refinement types to algebraic datatypes because data structures are often constructed or manipulated in such a way as to meet certain specifications—e.g., a list constructed by a sorting function must be sorted. A naive combination, however, leads to inefficient dynamic contract checking.

We propose *manifest datatypes*, of which argument types of data constructors can be refined, with the mechanism of casts between different but compatible datatypes to make dynamic contract checking more efficient. We formalize a contract calculus for manifest datatypes with a semantics and a type system and prove type soundness. We also describe systematic generation of manifest datatype definitions from refinement types, delayed contract checking, and a prototype implementation using Camlp4.

Gradual Effect Systems

Ronald Garcia, University of British Columbia, Canada

Effect systems have the potential to help software developers, but their practical adoption as part of language definitions has been very limited. We conjecture that this is due in part to the difficulty of transitioning from a system where effects are implicit and unrestricted to a system with a static effect discipline, which must settle for conservative checking in order to be decidable. To address this, we develop a theory of gradual effect checking, which makes it possible to incrementally annotate and statically check effects, while still rejecting statically effect-inconsistent programs. We extend the generic type-and-effect framework of Marino and Millstein with a notion of unknown effects, which turns out to be significantly more subtle than unknown types in traditional gradual typing.

This is joint work with Felipe Bañados Schwerter and Éric Tanter

Practical Gradual Typing for Dynamic OO Languages

Asumu Takikawa, Northeastern University, USA

Dynamically-typed OO languages have become a main staple of the practical programmer's toolkit. Once they build large systems in these languages, however, they realize that they want to equip their code with reliable type information for use in maintenance, documentation, and other software engineering tasks. While researchers have started to design gradual type systems to support these efforts, existing systems do not yet support the most dynamic features of these languages.

In this talk, I present an OO extension of Typed Racket. Like most dynamically typed OO languages, Racket takes classes seriously and supports them as first-class run-time values. As a result, Racket programmers rely on idioms such as mixins that require first-class classes. My talk will explain how Typed Racket's expanded type system accommodates these idioms and how its type-to-contract compiler ensures the soundness of the boundaries between typed and untyped components.

Joint work with: Daniel Feltey, Sam Tobin-Hochstadt, Matthias Felleisen

A calculus of contracting processes

Massimo Bartoletti, University of Cagliari, Italy

We address the problem of modelling and verifying contract-oriented systems, wherein distributed agents may advertise and stipulate contracts, but—differently from most other approaches to distributed agents—are not assumed to always behave "honestly". We model such systems in CO2, a basic calculus for contract-oriented computing. CO2 features primitives for advertising contracts, for reaching agreements, and for querying the fulfilment of contracts. Coordination among participants happens via multi-party sessions, which are created once agreements are reached.

The possibility of not keeping promises gives rise to a rich variety of possible misbehaviours, which we illustrate with the help some examples. We discuss some verification techniques to ensure " honesty ", namely that a participant always respects her contracts, in all possible execution contexts. The honesty property is quite strong, because it requires that a participant is capable of fulfilling her obligations also when the context plays against her. We discuss some work in progress about weaker notions of honesty (which however still guarantee safe interactions among agents), and their relation with different cases of inter-session dependencies. Our final goal is a theory for blame shifting, allowing to determine when a (not-strictly-honest) agent can blame other (dishonest) agents for her contractual violations.

A theory of agreements and protection

Massimo Bartoletti, University of Cagliari, Italy

We present a theory of contracts, interpreted as multi-player concurrent games among competitive participants. The two key notions of our model are that of agreement and protection. The agreement property is a game-theoretic generalization of the notion of compliance typically studied in session behaviours: a participant agrees on a contract if she has a strategy to reach her goals (or make another participant chargeable for a violation), whatever the moves of her adversaries. The protection property is relevant when contract brokers are untrusted: a participant is protected by a contract when she has a strategy to defend herself in all possible contexts, even in those where she has not reached an agreement.

We show that, in a relevant class of contracts, agreements and protection mutually exclude each other. We then propose a novel formalism for modelling contractual obligations (event structures with circular causality), which allows us to construct contracts which guarantee both agreements and protection. Finally, we establish a correspondence between game-theoretic contracts and Propositional Contract Logic, by showing that winning strategies in game-theoretic contracts are related to proofs in the logic.

Applications of Program Monitoring Tools for Software-Defined Networks

David Walker, Princeton University, USA

program monitoring techniques in the context of software-defined networks. In a software-defined network (SDN), a general-purpose controller machine decides on a global network policy and then transmits commands to many routers to implement that policy. A monitor for the SDN controller can sit in between the controller and the switches, monitoring the rules emitted from the controller and ensuring they establish some invariant (eg: no forwarding loops in the network; certain access control or reachability properties; etc).

Soft Contract Verification

David Van Horn, University of Maryland, USA

Behavioral software contracts are a widely used mechanism for governing the flow of values between components. However, run-time monitoring and enforcement of contracts imposes significant overhead and delays discovery of faulty components to run-time.

In this work, we tackle the problem of soft contract verification, which aims to statically prove either complete or partial contract correctness of components, written in an untyped, higher-order language with first-class contracts. Our approach uses higher-order symbolic execution that leverages contracts as a rich source of symbolic values, including unknown behavioral values, and employs an updatable heap of contract invariants to reason about flow-sensitive facts. The approach is able to analyze first-class contracts, recursive data structures, unknown functions, and control-flow-sensitive reasoning about values, which are all idiomatic in dynamic languages. It makes effective use of an off-the-shelf solver to decide problems without heavy encodings. The approach is competitive with a wide range of existing tools—including type systems, flow analyzers, and model checkers—on their own benchmarks.

Typed Racket's influence on contract systems

Sam Tobin-Hochstadt, Indiana University, USA

Typed Racket has always relied heavily on Racket's underlying contract system to maintain soundness when interacting with untyped programs. But as Typed Racket has grown more sophisticated, this has placed greater demands on the contract system. In this talk, I'll discuss several ways in which we've approached this question, and focus on one challenge in particular: interacting with encapsulated and mutable state. Our solution, called chaperones, enables Typed Racket to soundly exchange mutable data while not violating other invariants that the system relies on.

Session Types with Gradual Types

Peter Thiemann, University of Freiburg, Germany

Session types enable fine-grained static control over communication protocols. Gradual typing is a means to safely integrate statically and dynamically-typed program fragments.

We propose a calculus for synchronous functional two-party session types, augment this calculus with a dynamically-typed fragment as well as coercion operations between statically and dynamically-typed parts, and establish its basic metatheory: type preservation and progress.

We explore two different representations for coerced communication channels, with eager and lazy cast semantics. The lazy version is more efficient, but requires coercion simplification to establish the metatheory.

Higher-Order Temporal Contracts

Jay McCarthy, Brigham Young University, USA

Asynchronous message passing is an important paradigm in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms. Correctness in such an API is difficult to reason about manually, and testing against reference solutions is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control API internals to expose or reproduce errors. This paper provides a way to encode an MCAPI execution as a Satisfiability Modulo Theories (SMT) problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in a way that it now fails user provided assertions. The paper proves the problem is NP-complete. The encoding is useful for test, debug, and verification of MCAPI program execution. The novelty in the encoding is the direct use of match pairs (potential send and receive couplings). Match-pair encoding for MCAPI executions, when compared to other encoding strategies, is simpler to reason about, results in significantly fewer terms in the SMT problem, and captures feasible behaviors that are ignored in previously published techniques. Further, to our knowledge, this is the first SMT encoding that is able to run in infinite-buffer semantics, meaning the runtime has unlimited internal buffering as opposed to no internal buffering. Results demonstrate that the SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique, and it runs faster and uses less memory. As a result the encoding scales well for programs with high levels of non-determinism in how sends and receives may potentially match.

Contracts in Racket, from the Programmer's Perspective

Robby Findler, Northwestern University, USA

We will give a demo of Racket's contract system.

Sweet.js – JavaScript macros for contracts and security

Tim Disney, University of California, Santa Cruz, USA

Lisp and Scheme have demonstrated the power of macros to enable programmers to evolve and craft languages. In languages with more complex syntax, macros have had less success. In part, this has been due to the difficulty in building expressive hygienic macro systems for such languages. JavaScript in particular presents unique challenges for macro systems due to ambiguities in the lexing stage that force the JavaScript lexer and parser to be intertwined. In this paper we present a novel solution to the lexing ambiguity of JavaScript that enables us to cleanly separate the JavaScript lexer and parser by recording enough history during lexing to resolve ambiguities. We give an algorithm for this solution along with a proof that it does in fact correctly resolve ambiguities in the language. Though the algorithm and proof we present is specific to JavaScript, the general technique can be applied to other languages with ambiguous grammars. With lexer and parser separated, we then implement an expressive hygienic macro system for JavaScript called sweet.js.