TypeScript: The Next Generation

To boldly enforce types where no one has enforced types before

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Gavin Bierman
Microsoft Research
gmb@microsoft.com

Introduction

There is increasing interest in integrating dynamically and statically typed programming languages, as witnessed in industry by the development of the languages TypeScript and Dart, and in academia by the development of the theories of gradual types, hybrid types, and the blame calculus. The purpose of our project is to bring the academic and industrial developments together, applying theory to improve practice.

Our project focusses on JavaScript, an ECMA standard, and its typed variant TypeScript, an open-source project sponsored by Microsoft. JavaScript plays a central role in web-based applications and the new Windows 8 framework, and TypeScript is seeing rapid takeup, with over 150 JavaScript libraries now provided with TypeScript declarations. Our project has two parts, one aimed at immediate short-term application, and one aimed at fundamental long-term research.

1. *TypeScript TNG* The short-term goal is to build a tool, TypeScript: The Next Generation (or TypeScript TNG for short), that generates wrapper code from TypeScript `import` declarations to detect and pinpoint type errors. A wrapper will accept any JavaScript value as input, and either raise an error or return a value guaranteed to satisfy the invariant associated with the corresponding type. In particular, wrappers for generic types will assure surprisingly strong guarantees, known as "theorems for free". Our hypothesis is that TypeScript TNG will aid debugging and increase reliablility of TypeScript and JavaScript code.
2. *A wide-spectrum type system.* The long-term goal is to extend the foundations of the blame calculus to support a wide-spectrum of type systems, ranging from dynamic types (as in JavaScript or Racket) through generic types (as in F# or Haskell) to dependent types (as in F* or Coq). Our hypothesis is that a wide-spectrum type system will increase the utility of dependent types, by allowing dynamic checks to be used as a fallback when static validation is problematic.

The workplan below is likely to be too ambitious for a single PhD studentship. Which aspects are carried out will depend on which seem the most promising as our work develops, and on the abilities and desires of the student.

The next two sections detail the two parts of our research plan, and the final section summarises our track record.

1. TypeScript TNG

Types serve two different purposes in modern programming environments. The first is as a source of information to prompt the developer, for instance to populate a pulldown menu with methods that might be invoked at a given point. Indeed, providing effective prompts in Visual Studio is a primary motivation for the development of TypeScript. The second is as a source of correctness guarantees, for instance ensuring that a boolean is never passed where a number is expected. Ensuring correctness is a lesser motivation of TypeScript, because it is incompatible with another goal, supporting zero-cost interaction with JavaScript. However, recent developments in theory show how we might eat our cake and have it too: we can use wrappers to enforce type correctness guarantees while supporting low-cost interaction with JavaScript.

TypeScript allows the programmer to specify in an `interface` declaration types for a JavaScript module or library supplied by another party. The DefinitelyTyped repository (Yankov 2013) contains over 150 such declarations for a variety of popular JavaScript libraries. However, the information supplied by an `interface` is taken on faith. If, say, an interface declares that a callback expects a number when in fact the library supplies it a boolean, then an error may occur somewhere deep in the code of the callback (for instance, where the argument to the callback is used to index an array), or, worse, no error may be detected and a nonsensical result may be returned.

We intend to build a tool, TypeScript TNG, that generates wrapper code from TypeScript `import` declarations to detect and pinpoint type errors. A wrapper will accept any JavaScript value as input, and either raise an error or return a value guaranteed to satisfy the invariant associated with the corresponding type. As detailed in the next section, generation of wrappers will be based on the blame calculus; and, as detailed in the section after that, the wrappers for generic types ensure strong guarantees, known as “theorems for free”.

Adding wrappers leaves the behaviour of the code unchanged, so long as the library actually has the behaviour specified by the `interface` declaration, and so long as the library does not check for object identity on functions. We expect that most libraries do conform to suitable `interface` declarations and that use of object identity on functions is an uncommon corner case. Testing these hypotheses will be an important part of our research programme.

Wrappers are not free, but as they only impinge at the boundary between modules their cost is low. Wrappers are most important during the debug and test phases of a project, although (as with array bound checking) it is desirable to maintain them for production use as well. Measuring the runtime cost of wrappers will be another important part of our research programme.

Our theses are:

- It is possible to develop a tool to generate wrappers from `interface` declarations in TypeScript, such that the wrappers enforce the semantic properties expected by values of the given type, including the “theorems for free” associated with generic types.
- The tool will yield practical benefits: adding wrappers will aid in debugging and not impose undue run-time overhead. Most libraries will possess suitable `interface` declarations, for which adding wrappers will preserve the semantics.

The following sections review foundations on which our proposal is built, the blame calculus and Theorems for Free, review related work on F*, JS*, and TS*, and ResearchTS, and details our plans for evaluation.

Blame calculus The long tradition of work that integrates static and dynamic types includes the *partial types* of Thatte (1988), the *dynamic type* of Abadi et al. (1991), the *coercions* of Henglein (1994), the *contracts* of Findler and Felleisen (2002), the *dynamic dependent types* of Ou et al. (2004), the *hybrid types* of Gronski et al. (2006), the *gradual types* of Siek and Taha (2006), the *migratory types* of Tobin-Hochstadt and Felleisen (2006), and the *multi-language* programming of Matthews and Findler (2007). Integration of static and dynamic types is a feature of .NET languages including C# and Visual Basic, is being explored for JavaScript, Perl, Python, and Ruby, and is the subject of the recent series of STOP workshops.

The foundation for our proposal is the blame calculus, a core calculus that supports integration of a variety of type systems. The origin of the blame calculus lies in the observation that two research papers published in the same year used the same techniques to different ends. The gradual types of Siek and Taha (2006) integrate dynamic types with static types, while the hybrid types of Flanagan (2006) integrate generic types with refinement types. Both use essentially the same source language (lambda calculus), the same intermediate language (lambda calculus with casts), and the same type-directed translation between the two. Both are based on the contracts of Findler and Felleisen (2002), but neither use the positive and negative labels introduced by that calculus, which are essential to allocating blame to either the context containing the cast or the term contained in the cast.

The key result about blame was to observe that when casting between two types, if one is more precise than the other then blame always lies on the less-precisely typed side of the cast; we refer to this result as the blame theorem. Because gradual types and hybrid types lack positive and negative blame labels, the works cited above could not even begin to formulate such a result. Results similar to the blame theorem had been established by Tobin-Hochstadt and Felleisen (2006) and Matthews and Findler (2007), but each required a complicated proof based on operational equivalence. Wadler and Findler (2009) established the first simple proof of the blame theorem, using the traditional technique for proofs of type soundness based on progress and preservation (Wright and Felleisen 1994).

One important use case for blame is as follows. Consider the situation mentioned above, where a module written in TypeScript imports a library written in JavaScript, specifying the types of the library using an `interface` declaration. As mentioned above, currently in TypeScript the types in the `interface` declaration are taken on faith. Because JavaScript is untyped, there is no guarantee that a value passed from the JavaScript module will correspond to the `interface` declaration. Further, because TypeScript is not designed to provide watertight type soundness, there is no guarantee for a value passed from the TypeScript module either. Interposing a wrapper derived from a cast in the blame calculus would allow any violations of the `interface` declaration to be pinpointed, with blame corresponding to an indication of *which* of the two modules is at fault. In the presence of higher-order functions it may not be completely trivial to decide which module is in error using informal reasoning—the blame calculus provides a simple formalism for accurately allocating blame. The technique works not just for a single import, but extends to import of multiple modules written in JavaScript, each passing objects and functions to the other, so long as each has types provided by an `inferface` declaration.

Theorems for Free Ahmed et al. (2011) extended the gradual typing fragment of the blame calculus to include polymorphic types, which correspond to the generic types found in Java and C#, and recently added to TypeScript. A fundamental semantic property of polymorphic types is *relational parametricity*, as introduced by Reynolds (1983) and popularised under the name “theorems for free” by Wadler (1989). A statically typed value of generic type is guaranteed to satisfy certain properties.

For instance, in a statically-typed language without side effects, a value of type $\forall$`T. T` $\rightarrow$ `T` must either be the identity function (which always returns its argument) or the undefined function (which never returns a value). The intuitive reason for this is that the function must work for *any* type `T`, while examining a value or generating a new value requires knowing something about the structure of its type. Similarly, a value of type $\forall$`T. List<T>` $\rightarrow$ `List<T>` must be a rearranging function, such as one that reverses a list or drops its first element; it cannot examine the value of the elements passed to it or generate new elements to return. In the presence of side effects, similar though slightly weaker properties still hold.

What is remarkable is that the wrappers generated by the blame calculus can guarantee this property, even though the value passed into the wrapper is from a dynamically-typed language and satisfies no constraints whatsoever. This is achieved by use of sealing, closely related to cryptographic sealing. The required operation is to be able to generate a pair of functions, one which seals a value and one which unseals the value. A sealed value is opaque and cannot be examined save by applying the corresponding unsealing function, which returns the original value. Wrappers for generic types use the types as a guide to sealing. When a wrapper encounters an instantiation of a generic function, it generates a new pair of sealing and unsealing functions; only the wrapper has access to these functions. Each value of generic type is sealed by the wrapper when it is passed into the function, and unsealed when it is returned.

Hence, in the case of a function of type $\forall$`T. T` $\rightarrow$ `T`, if a value is returned other than the argument it will not be properly sealed, and the unseal function applied to the result will raise an error. Similarly, in the case of a function of type $\forall$`T. List<T>` $\rightarrow$ `List<T>`, the seal prevents elements of the input list from being examined, and ensures that the only elements in the output list must come from the input list. As with blame, in the presence of higher-order functions it may not be completely trivial to decide which values to seal or unseal using informal reasoning—the extended blame calculus provides a simple formalism that specifies when sealing and unsealing is required.

Relational parametricity underlies some program optimizations, notably *shortcut deforestation* as employed by the Glasgow Haskell Compiler (Gill et al. 1993). Our system guarantees the validity of such optimizations even in the presence of dynamic types.

F*, JS*, and TS* F* is a new, dependently typed language for secure distributed programming. It is designed to enable the construction and communication of proofs of program properties and of properties of a program's environment in a verifiably secure way. F* compiles to .NET bytecode in type-preserving style, and interoperates smoothly with other .NET languages, including F#, on which it is based. F* subsumes several prior languages, including Fine, F7, FX, and others. It has been used to verify nearly 50,000 lines of code, ranging from crypto protocol implementations to web browser extensions, and from cloud-hosted web applications to key parts of the F* compiler itself. The F* language is described in Swamy et al. (2011a), and its use to certify its own compiler is described in Strub et al. (2012). [Parts of this paragraph are taken from the F* web site.]

Recent work has examined the use of F* to generate JavaScript satisfying strong security properties. JavaScript code that interacts with third-party code is subject to a number of security attacks, such as modifying the prototype of an object to change the object's behaviour, or traversing stack frames to examine the internal state of a function.

Fournet et al. (2013) introduce JS*, which compiles a large subset of F* to JavaScript. The compilation is fully abstract, meaning that if two pieces of F* code behave identically in any F* context, the versions compiled by JS* will behave identically in any JavaScript context. This is a surprisingly strong property, given that the JavaScript environment can break a number of abstractions assumed in F*, for instance using the attacks on prototypes or stack frames mentioned above. The JS* compiler makes clever use of wrappers to prevent such attacks. The compiler is written in F*, and has been proved fully abstract within F* itself—a remarkable achievement.

Swamy et al. (2013) goes one step further, and introduces TS*, a variant of TypeScript which supports writing web programs that are guaranteed not to be subject to the security attacks mentioned above. The technique is to translate TS* to F*, and then to translate F* back to JavaScript with the fully-abstract JS* compiler. The wrappers that enforce full abstraction guarantee freedom from a range of attacks, and hence yield code satisfying strong security properties. An interesting aspect of the system is the use of two distinct types, `Any` and `Un`, to separate values known to satisfy security properties (of type `Any`) from those that may contain attacker code (of type `Un`).

The wrappers used in JS* and TS* are closely related to those used in the blame calculus, leading to several fruitful lines for investigation. In one direction, one might adapt blame tracking and the use of sealing to handle generic types from the blame calculus to JS* and TS*. In the reverse direction, one might adapt the methods of preventing security attacks from JS* and TS* to the blame calculus.

ResearchTS Bierman is currently collaborating with the TypeScript product team on the design of the TypeScript type system. This ongoing process has already led to several changes in the design, implementation, and public specification of TypeScript, and some academic papers are in preparation. Additionally, a research variant of the TypeScript compiler, dubbed "ResearchTS", is currently being built. This variant of the compiler allows for various experiments on the type system. Currently compiler options are being added to restrict the type system to ensure various levels of static safety, and also to guarantee specific security properties of the emitted JavaScript.

ResearchTS is slated for open-source release and will provide a convenient platform for the implementation work described in this proposal.

Evaluation Once a wrapper generator has been built on top of ResearchTS, we will take steps to evaluate its efficacy. Our hypotheses are as follows.

- JavaScript TNG will reduce debugging time.
- JavaScript TNG will impose only modest runtime costs.
- JavaScript TNG will preserve the semantics of most libraries: adding wrappers will yield the same results.
- Corner cases where adding JavaScript TNG wrappers change the semantics, such as testing for pointer equality on functions, will be rare.

We will measure to determine whether each of these is the case. Test cases can be taken from the DefinitelyTyped repository (Yankov 2013), which already contains over 150 declarations for JavaScript libraries.

Dependent types As types become more powerful, the wrappers generated by TypeScript TNG enforce stronger invariants. A next step in the work described here might be to extend TypeScript to support dependent types. Such work requires a foundational study of how to integrate dynamic types and dependent types, which is the study of the next section.

2. A wide-spectrum type system

Politicians often claim that our differences make us stronger; our aim here is to apply that truism to the field of programming languages.

Programming languages offer a range of type structures. Here are four of the most important, listed from weakest to strongest:

- Purely dynamic, as in Racket, Python, and JavaScript
- Generic types, as in ML, Haskell, and F#
- Refinement types, as in Dependent ML and F7
- Dependent types, as in Coq, Agda, and F*

The different systems provide different levels of guarantee. A dynamically typed program may fail by, for instance, supplying an integer where an array is expected. A generic typed program will guarantee to supply an array where an array is expected, but may fail by providing an array index outside of the array bounds; guarantees are provided by unification-based type inference. A refinement typed program may guarantee that array indices are always within bounds, but may fail by entering an infinite loop; guarantees are provided by applying an SMT solver. A dependently typed program will guarantee to never raise exceptions or enter an infinite loop; guarantees are provided by user-supplied proofs. What we call "refinement types" is sometimes called "weak dependent typing", and what we call "dependent types" is sometimes called "strong dependent typing"; they correspond to two different kinds available in F*.

Recent work on contracts has suggested the use of checked or validated casts to interface between such systems. For instance, one can cast a value of dynamic type to a generic type, or a value of generic type to a refinement type. Such casts may require conditions to be satisfied. (Is this dynamic value an array? Is this integer within the array bounds?) Such tests may be checked dynamically at runtime, or validated statically at compile time, in the latter case via either an SMT solver or a user-supplied proof. Contracts have a sound theory, and are particularly important when dealing with higher-order functions.

Recent work on effect systems suggest that one can use effect types to classify what guarantees are provided in what segments of code. For instance, effects might record whether or not a code segment contains a cast that is dynamically checked, or whether or not a code segment may enter an infinite loop. Proofs may be represented by programs, so long as the program contains no dynamic checks and cannot loop.

Modern systems partake of the entire range of type systems listed above. Browsers and the Windows 8 framework depend heavily on JavaScript, while advanced systems for security build on refinement types and dependent types. We believe that a system integrating all the typing styles would be stronger than the sum of its parts. For instance, one set of developers could write JavaScript applications on top of a security layer validated by a different set of developers, with dynamically checked tests (similar to the wrappers generated by TypeScript TNG) ensuring that the preconditions assumed by the security code are met by the JavaScript applications.

Our theses are:

- It is possible to develop a wide-spectrum type system covering all the above disciplines, building on the ideas of the blame calculus and effect systems.
- A wide-spectrum system will increase the utility of the different styles of typing, by allowing dynamic checks to be used as a fallback when static validation is problematic.

The following sections detail specific areas requiring development: theoretical work on extended foundations for the blame calculus and on effect systems, an implementation with case studies, and possible further work on property-based testing.

Blame calculus foundations The first paper on the blame calculus supports base types, function types, refinements over base types, and type dynamic (Wadler and Findler 2009). The second paper adds polymorphism, but removes refinements over base types (Ahmed et al. 2011). Two extensions are required as a basis for further work.

First is to add refinements at all types (not restricted to base types), and to add dependent types. Refinements at other than base type pose technical difficulties, but recent work introduces a new technique that appears to avoid the problems (Belo et al. 2011). We expect to be able to base our work on this new approach, the interesting new step being to add the type dynamic. The adaptation will require dealing with several subtle differences: Belo et al. (2011) requires a cast from a polymorhic type to be to a polymorphic type (and conversely), but permits the body of a polymorphic abstraction to be any term, while Ahmed et al. (2011) permits casts from polymorphic type to any compatible type (and conversely), but requires the body of a polymorphic abstraction to be a value. Once the extended framework is established, one must suitably extend the definitions of the four subtyping relations, and establish the equivalent of the blame theorem for the extended system.

Second is to investigate the tradeoffs between lazy and eager approaches to contract enforcement. Each of these is known to have advantages but also to entail costs (Degen et al. 2010; Findler et al. 2007; Hinze et al. 2006; Swamy et al. 2013). The tradeoffs are less important when dealing with functions, but are significant when dealing with data structures such as tuples and lists. It would be useful to develop both lazy and eager variants of the blame calculus to understand the tradeoff between them, and to be able to choose the appropriate variant when developing programs.

If the student has a strong theoretical bent, additional work in this area could define and prove parametricity properties by establishing a suitable step-indexed logical relation.

Effect systems Programs that use dynamic checking may raise blame at runtime, and programs with unbounded recursion may loop forever. It is desirable to permit these computational effects in program code, while sound representation of proofs as code requires banning such effects. We propose to track the presence or absence of such effects using an effect type system, of the kind introduced by Gifford and Lucassen (1986).

In previous work, we related effect types to monads (Wadler and Thiemann 2003). A practical difficulty with effect types is that they require labelling every function type with an effect, which may render types unwieldy. Recent work on type systems for monads by Swamy and Leijen at MSR and Guts and Hicks at Maryland restricts the places at which effect labels are required while maintaining type inference and coherence, and may be useful in ensuring effect types remain tractable (Swamy et al. 2011b).

The problem of permitting exceptions and loops in general code while prohibiting them in code used to represent proofs also occurs in the $F^\star$ system (Swamy et al. 2011a), where it is solved by an innovative use of kinds. We would like to explore and understand the relation between using effects and using kinds to delimit code used for computation from code used for proofs.

Implementation and case studies Apart from the theoretical foundations outlined above, it is desirable to design and implement a practical system for writing such programs, and attempt suitable case studies. Implementing a new system from scratch is probably not desirable; suitable base systems for the work include F# and $F^\star$. There is already a suite of case studies for $F^\star$, including a cloud application managing a medical record database, a curated database with provenance recorded as proof terms, a suite of seventeen browser extensions, and a cryptographic library used to implement two- and three-party sessions (Swamy et al. 2011a). One possibility is to adapt these studies, paying particular attention to the tradeoff between run-time checking and compile-time validation. We conjecture that usability will be significantly increased by permitting run-time checking as a workaround when compile-time validation is infeasible.

Property-based testing Property-based testing tools such as QuickCheck have attracted considerable interest as an alternative to unit testing for increasing the reliability of systems (Claessen and Hughes 2000). Over the last decade, these tools have seen increased use by software developers such as Ericsson, the appearance of commercial suppliers such as Quviq, and the founding of an EU funded project (Derrick et al. 2009).

Refinement and dependent types in our system may serve as a source of properties for testing, using techniques similar to those developed for QuickCheck. From the declaration of each type one can automatically derive a random generator of values of that type, and then check that the properties specified by the refinement and dependent types hold. This complements run-time checking and compile-time validation: whereas run-time checking tests that the specified properties hold for each invocation of a function during program execution, and compile-time validation proves that the properties always hold, property-based testing checks that the properties hold for a collection of randomly generated values, and searches for a simple counterexample if a test fails. It may be possible to either use property-based testing as a separate stand-alone test separate from program execution, or in combination with program execution. In particular, the technique of run-time testing cannot easily apply to properties that include quantifiers, but such properties are a perfect candidate for property-based testing.

3. Track record

Philip Wadler is Professor of Theoretical Computer Science in the School of Informatics of the University of Edinburgh. His career spans academia and industry, with degrees from Stanford and Carnegie-Mellon, posts at Oxford, Glasgow, Bell Labs, and Avaya Labs, and guest professorships in Sydney, Copenhagen, and Paris. He contributed to the design of generics for Java, to the W3C standard XQuery, and to the functional language Haskell, and is Principal Investigator on a five-year £4M EPSRC programme grant, “From Data Types to Session Types: A Basis for Concurrency and Distribution”. He co-founded the *Journal of Functional Programming*, served as program chair for ICFP and POPL, and is Past Chair of ACM SIGPLAN. He is a Fellow of the ACM, a Fellow of the Royal Society of Edinburgh, and a former Royal Society Wolfson Merit Fellow. He appears second in the list of “Top Programming Language Researchers” on Microsoft Academic Search, and has an h-index of 60 on Google Scholar.

Gavin Bierman is a senior researcher at Microsoft Research in Cambridge. He holds degrees from Imperial and Cambridge. Before joining Microsoft he was a University Lecturer at the University of Cambridge. At Microsoft his research has focused on type systems, semantics, separation logic, database query languages and dynamic software updating and he has made contributions to several production languages including C#, Visual Basic, and TypeScript.

University of Edinburgh The School of Informatics has top ratings for both teaching and research. Wadler sits in the Laboratory for the Foundation of Computer Science, with a long tradition of high-quality research linking theory to application. The productive relation between Microsoft and Edinburgh is recognised by the University of Edinburgh Microsoft Research Joint Initiative in Informatics.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268, 1991.

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011.

J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, pages 18–37, 2011.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

M. Degen, P. Thiemann, and S. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *J. Log. Algebr. Program.*, 79(7):515–549, 2010.

J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-Å. Fredlund, V. M. Gulías, J. Hughes, and S. J. Thompson. Property-based testing—the ProTest project. In *FMCO*, pages 250–271, 2009.

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

R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *IFL*, Oct. 2007.

C. Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, 2006.

C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL*, pages 371–384, 2013.

D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Lisp and Functional Programming*, 1986.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme Workshop*, 2006.

F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Programming*, 22(3):197–230, 1994.

R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *FLOPS*, volume 3945 of *LNCS*, pages 208–225, 2006.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, 2007.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *Conference on Theoretical Computer Science*, 2004.

J. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing*, pages 513–523. North-Holland, 1983.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme Workshop*, 2006.

P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: bootstrapping certified typecheckers in $F^\star$ with Coq. In *POPL*, pages 571–584, 2012.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011a.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011b.

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript, 2013. Draft paper.

S. Thatte. Type inference with partial types. In *ICALP*, volume 317 of *LNCS*. Springer-Verlag, 1988.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

P. Wadler. Theorems for free. In *FPCA*, 1989.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP*, pages 1–16, 2009.

P. Wadler and P. Thiemann. The marriage of effects and monads. *TOCL*, 4(1):1–32, 2003.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

B. Yankov. Definitely typed repository, 2013. `https://github.com/borisyankov/DefinitelyTyped`.