Static Types with Less Syntax: Locus Types

by

Adam Domurad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Optionally and gradually-typed languages allow types to be introduced to dynamic code as needed. While this approach allows some gradual movement from dynamically to statically-typed code, it requires rewriting object-constructing code to use conventional static types. We introduce a flexible notion of type, deemed "locus types", that aims to minimize syntactic burden and the need for refactoring when introducing types to dynamic code. Locus types are gained by objects that pass through an annotated code site, following the creed of "code is types". Their structure is inferred from local type information computed through flow-based type refinement. The design of LocusTypeScript, a language extending TypeScript with locus types, is detailed. Tooling support, building on that of TypeScript, for programming with locus types is described. As well, the general properties and applicability of locus types are explored. LocusTypeScript's simple algorithm for computing stable flow-based refinement types is presented. The implications and performance impact of making locus types sound are discussed.

Acknowledgements

On the technical side of things, I make the following acknowledgements:

My supervisor, Gregor Richards, for conceiving the idea of locus types and playing an active role in their design, as well as his insight into many areas of computer science.

The TypeScript team, for their excellent engineering practices that aided the derivative implementation described herein, especially with regard to maintaining simplicity.

Pat Goebel, for his insight and peer review.

Ifaz Kabir, for testing LocusTypeScript in practice and his insight into the corner cases of the compiler.

On the less technical side of things, I make the following acknowledgements:

To Gregor Richards, for his patient guidance and his candid nature that made our conversations both productive and amusing.

To Pat Goebel, for his seemingly universal enthusiasm and competence.

To Matt Simon and Tommy MacLean, for being as much family to me as they were friends.

To my family, for being as much friends to me as they were family.

And to my dog Lola, for her unconditional willingness to go for walks at odd hours.

Table of Contents

List of Figures ..... **vii**

1 Introduction **1**
- 1.1 JavaScript, TypeScript, StrongScript 2
- 1.2 Optional and gradual typing 2
- 1.3 Flow-based type refinement 3
- 1.4 Locus Types 3
- 1.5 LocusTypeScript 4
- 1.6 Thesis Outline 5

2 Background **6**
- 2.1 JavaScript 6
- 2.2 TypeScript 7
- 2.3 StrongScript 8
- 2.4 Gradual typing 9
- 2.5 Flow-based type refinement 10
- 2.6 Related work 11

3 Motivation **12**

4 Locus Types and LocusTypeScript **15**
- 4.1 Overview 15
- 4.2 Deriving Locus types 17
- 4.3 Handling interprocedural flow 19
- 4.4 Type granularity 21
- 4.5 Soundness 22
- 4.6 Nominality and checkability 22
- 4.7 Interaction with gradual types 23

4.8 Run-time protection 24
4.9 Backwards compatiblity 26
4.10 Tooling 27

5 Formal Properties 29
5.1 Core semantics 29
5.2 Metatheory 34
5.3 Imperative extensions 37

6 Locus Type Elaboration 40
6.1 Language abstraction 40
6.2 Overview 41
6.3 Internal analysis state 42
6.4 Algorithm 43
6.5 Proof of properties 43
6.6 Extension to full objects 46
6.7 Handling recursive types 46
6.8 Variations and extensions 47

7 Performance 49
7.1 Performance 49

8 Software 51

9 Conclusions and Future Work 52

References 53

List of Figures

5.1 The type system 32
5.2 The run-time semantics 33
5.3 The imperative desugaring 38

6.1 Flow-based type refinement 44

7.1 Performance comparison of differing levels of enforcement. 50
7.2 Performance comparison on the V8 VM. Times are in milliseconds, lower is better. 50

Chapter 1

Introduction

As the scope of dynamically-typed languages increases dramatically past that of basic scripting, partial static type checking increasingly becomes an attractive tool to reduce the burden of software maintenance. While introducing mandatory static type checking is often impractical and detracts from the flexible nature of dynamically-typed languages, introducing partial static type checking presents design challenges not present with full static type checking. When a portion of a program is dynamically-typed, a strategy for handling the interaction between statically and dynamically-typed components is needed. This strategy leads to the distinction between *optionally* and *gradually*-typed languages. *Optionally-typed* languages allow for annotating a subset of a program with type annotations, which is statically checked up to interaction with unannotated code, with no impact on dynamic execution and thus performance. *Gradually-typed* languages similarly allow typing for a subset of the program, but provide stronger soundness guarantees, usually realized by run-time checks. Both optionally and gradually-typed languages allow for introducing types as required to a dynamic program, while optionally-typed languages do not emphasize soundness.

Parallel to this, systems using *flow-based type refinement* can determine type information in highly dynamic or otherwise unamenable languages, for purposes of type checking [9][10][12]. These systems observe statically how conditions and mutations in the code affect the possible types of a value, and in doing so, identify code points where types are used incorrectly, even without types being explicitly specified. Such type systems can describe quite precisely how values change within a constrained block of code, typically a single function.

Optional and gradual typing allow type specification, whether for program maintainability, or the stronger goal of type soundness, while type refinement allows for types to be inferred from existing, dynamic code. The goal of this thesis is to improve the convenience of introducing types to dynamically-typed code by using these

two tools in concert: Type refinement is used to determine the type of objects at a specified location in code, such as a constructor, and that type is made manifest like any named, explicitly defined type in the optional or gradually-typed language. This simplifies the task of introducing static type checking to dynamically-typed code and reduces its syntactic burden. We deem types introduced through this manner "locus types", and their annotation sites "locus type declarations".

The applicability of locus types to this goal is shown through the design and implementation of LocusTypeScript, an extension to the JavaScript language that allows for using locus types in either an optionally- or gradually-typed setting. LocusTypeScript builds on the TypeScript [13] and StrongScript [17] languages, which, respectively, provide frameworks for optional and gradual typing in JavaScript. Through locus types, types inferred locally by flow-based type refinement are reified into globally usable named types.

These types are then used identically to the types already in TypeScript and StrongScript.

1.1 JavaScript, TypeScript, StrongScript

As the primary language available to program web applications, JavaScript has received a great deal of attention towards managing programs at ever-increasing scale. Part of this effort are a number of derivative languages that add support for partial static typing, aiming to ease the burden of testing large dynamically-typed programs. Industrial languages such as Microsoft's TypeScript [13] and Facebook's Flow [5] have introduced optional types to the popular dynamic language JavaScript, forming derivative optionally-typed languages. StrongScript [17] is a derivative of TypeScript which adds optional soundness, enabling gradual typing. While TypeScript and StrongScript adopt a largely traditional type system, Flow additionally features flow-based type refinement to capture precise type information without the need for explicit types.

1.2 Optional and gradual typing

Optional type systems were introduced with Strongtalk [4], which added type annotations to Smalltalk. The philosophy behind Strongtalk is mirrored in TypeScript: Optional types have no effect on the program's run-time semantics [3], and are never mandatory. These optional type systems are purely type-erased (i.e., types have no effect on semanatics), and highlight the impact of pure type erasure. Any program which uses types everywhere—as would be mandatory in a purely static type

system—is itself sound, and thus any unsoundness is caused by components of the program which lack types.

The term "gradual typing" arose from Siek and Taha [18][19], who used it to refer to languages in which type annotations can be added gradually to untyped code. In their setting, like with software contracts, wrappers are used to enforce types. LocusTypeScript builds on the gradually-typed language StrongScript, which includes optional typing features. LocusTypeScript uses StrongScript's type enforcement capabilities to implement locus types with opt-in soundness.

1.3 Flow-based type refinement

Parallel but unrelated to gradual typing, various systems have been used to determine statically the run-time types of values in dynamically-typed languages. Using information flow analysis to conservatively estimate types yields the type refinement technique [10][12]. Type refinement takes advantage of information flow to make precise statements about the types of values at particular points in the program.

This work uses type refinement as a means of inferring the definition of types in a gradually-typed system, encapsulating local type information in globally usable types. Since we aim to capture general types, we do not require (or desire) the full depth of type refinement's analysis. The light type refinement presented infers fairly precise types using only data-flow analysis local to a specific function or block.

1.4 Locus Types

Locus types are a form of type annotation used to give a name to a type discovered by flow-based type refinement, thereby turning the site at which an object is created or extended into a type declaration. A variable that is subject to type refinement is annotated with a locus type declaration, which gives an explicit name to the refined type. This explicit name can then be used globally to refer to this type, just like a typical type in an optionally- or gradually-typed language. In essence, a locus type declaration determines the structure of the type implicitly from how an exemplary instance is used, allowing the programmer to specify nothing more than the type's name:

```
// Simple example, 'Point' type introduced based on context:
var point: declare Point = {x: 1, y: 1};
```

Outside the scope of their declaration, locus types are used exactly like explicitly declared types; the only difference is that their declaration is not explicit. Within

the scope of their declaration, a variable annotated with a locus type declaration can be modified like an untyped object. This capacity gives the power of dynamic languages where it is needed, in creation, and the static guarantees of optional or gradual typing where they are needed, in use. Locus types allow programmers to use type annotations without explicitly declaring the structure of the objects in their programs.

1.5 LocusTypeScript

LocusTypeScript, the implementation of locus types presented herein, is an extension to StrongScript [17], which is in turn an extension of TypeScript. The ideas, however, are valid for any optionally- or gradually-typed language in which objects may be extended at run-time. Locus types can complement any gradually-typed language of this form, and primarily offer a new way of specifying types, not a semantically-distinct feature. Introducing explicit types in other optionally- or gradually-typed languages often requires explicitly noting details already known to the type system. Consider, for instance, the simple snippet from above which creates a point object in JavaScript:

```
// No type given, fully dynamic semantics:
var point = {x: 1, y: 1};
```

As was shown above, in LocusTypeScript, it is enough to add a single type annotation to introduce a globally usable `Point` locus type:

```
var point: declare Point = {x: 1, y: 1};
```

This is equivalent to writing an explicit `Point` type with `x` and `y` `number`-typed members, but syntactically more brief. In essence, `declare` exposes the type information inferred into a globally usable type. This generalizes to more sophisticated declarations, reducing the burden of both writing type annotations and keeping them in sync with declaring code. Further, when the programmer is satisfied that the type isn't going to change, IDE tools allow them to convert the implicit locus type to an explicit interface, promoting a smooth transition to more stable types.

TypeScript's tooling API was extended for LocusTypeScript to provide support for refactoring with locus types, realized in an Atom plugin building on `atom-typescript`. The refactoring techniques readily generalize to other languages, as well.

1.6 Thesis Outline

The goal of this work is to present a scheme for using type information uncovered from flow-based type refinement techniques to reduce the syntactic burden of introducing static types to dynamically-typed programs. The contributions of this work are a broadly applicable design for language annotations that specify explicit names for otherwise implicitly derived types, a JavaScript-compatible implementation of such a language based on TypeScript, and a brief exploration of tooling for locus types. As well, this work presents a system of integrating flow-based type refinement into a gradually-typed language amenable to run-time enforced soundness. The performance impact of applying this protection scheme to emitted JavaScript is explored.

Specifically, the following contributions are made:

- Introduce the new concept of "locus types", a widely applicable scheme for using flow-based type refinement as an evolutionary step towards, or light-weight replacement for, conventionally specified types.
- Explore and evaluate how locus types interact with soundness.
- Design and implementation of LocusTypeScript, applying locus types in a JavaScript setting (through extensions to TypeScript).
- Demonstrate how locus types can be used by IDE's, through tool support for LocusTypeScript, allowing smooth movement from locus types to conventional types.

The contents of this thesis are as follows:

Chapter 2 presents additional details about background concepts related to locus types and LocusTypeScript.

Chapter 3 presents motivating examples of cases where locus types fit particularly naturally when introducing types to code.

Chapter 4 details the design and features of LocusTypeScript.

Chapter 5 presents the relevant formal details of locus types in a gradual typing setting with respect to λ_{JS}, a formalization of JavaScript.

Chapter 6 presents the locus type derivation algorithm, and some of its properties.

The algorithm is presented in a general form that applies to any imperative language.

Chapter 7 presents an evaluation of the performance of enforcing soundness in LocusTypeScript.

Chapter 8 briefly describes the software behind LocusTypeScript, and where it can be obtained.

Chapter 2

Background

2.1 JavaScript

JavaScript is an imperative, object-oriented dynamically typed programming language made popular as "the language of the web" due to its inclusion in web browsers. JavaScript lacks type annotations, but is otherwise syntactically similar to Java and other languages in the C family:

```
function adder(x, y) {
    return x.value + y.value;
}
```

The object model underlying JavaScript is based on Self's [23]. Objects in JavaScript may be specified with constructors, granting them prototypes that create a dynamic class-like hierarchy, or may be specified as object literals, which inherit only from the root of the inheritance hierarchy, `Object.prototype`. Function-valued fields act as methods when called after access, with an implicit `this` parameter provided. These two features can be used to create objects that behave similarly to classes:

```
function Circle(radius) {
    this.radius = radius;
}
// Implicitly passed 'this' when called:
Circle.prototype.area = function() {
    return Math.PI * this.radius * this.radius;
}
var c = new Circle(1);
print(c.area()); // => 3.141592...
```

This mechanism allows a highly dynamic form of inheritance and overriding.

Objects may also be specified as object literals, which inherit only from the root of the inheritance hierarchy, `Object.prototype`.

Fields may be added to objects or their prototypes at any time, with the consequence that it can be difficult to predict the API of an arbitrary object at run-time. JavaScript hides much of the details of field membership through a special value, `undefined`. Undefined fields evaluate to this value rather than producing an error.

```
var c = new Circle(1);
print(c.notFound); // => undefined
```

Several other behaviors that would traditionally be considered type errors also evaluate to `undefined`. As a consequence, JavaScript rarely outright fails at run-time, but often produces erroneous or nonsensical results.

2.2 TypeScript

Microsoft's TypeScript extends JavaScript by adding optional types. It does so with full type-erasure, and allows values to flow from untyped to typed expressions with no run-time checks. As such, it is unsound, but its type system has no impact on performance relative to JavaScript. In TypeScript, the `Circle` example above would be written as a class:

```
class Circle {
    constructor(public radius: number) {} // assigns member
    public area() {
        return Math.PI * this.radius * this.radius;
    }
}
// Usage and semantics remain the same:
var c = new Circle(1);
print(c.area()); // => 3.141592...
```

Parameters to the constructor are treated specially. In the above instance, `radius` is a field of `Circle` with type `number`. Parameters not marked `public` (or `private`) do not become fields, and fields may be specified outside the constructor's parameter list.

Any misuse of the Circle type detectable at compile time raises a type error:

```
var c = new Circle(1);
print(c.notFound); // compile-time error
new Circle("Foo"); // compile-time error
```

However, types can be ignored, and the underlying semantics is still JavaScript:

```
// cast away types, no error
var c = <any> new Circle(<any> "Foo");
print(c.notFound); // prints "undefined"
```

TypeScript's types are structural, and so an object need not, e.g., be an instance of the `Circle` class in order to satisfy the `Circle` type:

```
function circumference(c: Circle) {
    return 2 * Math.PI * c.radius;
}
// object literal compatible with Circle type,
// no error at compile-time or run-time:
circumference({radius: 1, area: function() {
    return Math.PI;
}});
```

The core of TypeScript's type system is sound [1]. Unsoundness arises from the boundary between typed and untyped code, and downcasts, which are unchecked. Even without soundness, however, optional types are useful for catching a class of errors at compile time, and for software engineering concerns such as autocompletion in development environments. This guarantee is similar to gradually-typed languages, but TypeScript allows programs to continue with types violated due to unsoundness, where a gradually-typed system would reject them at run-time.

2.3 StrongScript

StrongScript [17] is a derivative of TypeScript that adds optional soundness. While TypeScript never enforces any types at run-time, StrongScript adds a "concrete" type constructor "`!`", which, when added to a suitable type, enables run-time enforcement. LocusTypeScript uses StrongScript and its "`!`" type constructor so that locus types can also benefit from run-time enforcement, but is otherwise an orthogonal extension to TypeScript. For instance, one can enforce that `Circle`'s have numeric radiuses by changing their constructor as follows:

```
    constructor(public radius: !number) {}
```

With `!number` replacing `number`, any attempt to violate the contract that `Circle`'s have numeric radiuses will result in an error:

```
c = new Circle(1); // OK, verified at compile-time
c = new Circle(<any> 1); // OK, verified at run-time
c = new Circle(<any> "Foo"); // error at run-time
c.radius = 1; // OK, verified at compile-time
```

```
c.radius = <any> 1; // OK, verified at run-time
c.radius = <any> "Foo"; // error at run-time
```

To enable eager run-time enforcement, StrongScript introduces nominal types to TypeScript, which otherwise uses structural typing exclusively. These nominal types have intrinsic identity, and any subtypes are declared explicitly. In order for a value to be typed as a `Circle`, it must have been created by `new Circle`. This strictness allows for classes to be concretely typed:

```
function circumference(c: !Circle) {
    return 2 * Math.PI * c.radius;
}
// fails at run-time with cast (or statically without):
circumference(<any> {radius: 1, area: function() {
    return Math.PI;
}});
```

To allow for certain run-time checks, StrongScript's types are nominal, unlike TypeScript's. LocusTypeScript inherits this change to TypeScript, and so LocusTypeScript's locus types are nominal as well. The concept of locus types is not tied to nominality, however. The impacts of nominal typing on locus types, and how locus types can be implemented in a structural setting, are discussed in Section 4.7.

2.4 Gradual typing

StrongScript and TypeScript are, to different degrees, examples of gradually-typed programming languages. They did not introduce the concept, however, and differ in crucial ways from other gradually-typed languages.

Gradual typing itself relates to research on software contracts [6]. Contracts allow the enforcement of constraints considerably more rich and semantically meaningful than types (e.g., enforcing arbitrary relationships between object components). Indeed, through wrappers [7], even contracts for higher-order functions are possible, as wrappers can validate pre- and post-conditions. Wrappers preserve invariants by adding layers of indirection that check arbitrary conditions on objects while otherwise preserving object behaviour. *Typed Racket* exemplifies later work in this line of research [21][22].

These two related lines of research have cemented many ideas of gradual typing, in particular how blame (i.e., which part of code is at fault) may be assigned in a system using wrappers to enforce contracts [26]. Both approaches do have the same significant drawback, however: Wrappers impose a significant performance penalty [20]. Various solutions have been investigated, but significant slowdowns are still reported.

An alternative approach has been to eschew higher-order contracts and define the underlying type system such that types can always be checked immediately, with no wrappers required. Typically this is done by using nominal types. This line of work began in the Thorn [2] language. As nominal type systems are less flexible than structural and higher-order type systems, several systems in this vein have adopted "like-types" [27]. Like types optionally remove contracts, and are thus equivalent to optional types: Erased and unsound. Systems with like types allow sound but inflexible nominal types, unsound but flexible like types, and of course untyped code. StrongScript, which LocusTypeScript builds upon, is one such system.

2.5 Flow-based type refinement

The type refinement technique allows for extracting type information from program control flow. As an example, consider attempting to assign a static type to `x`:

```
x = {radius: 1};
```

It is fairly obvious that `x` can be typed as an object, and `x.radius` can be typed as a number. In fact, `x.radius` could (hypothetically) be typed as the value `1`.

Information flow affects types when control flow occurs. For instance, a naïve type checker may give `y` the type `boolean` ∪ `string` in this example:

```
if (x.radius === 1) {
    y = true;
} else {
    y = "whoops";
}
```

However, this analysis can be refined to `boolean` by observing the type-related effects of the conditional. The second branch is unreachable, and so `y` cannot be a string.

In the absence of whole-program analysis, which is often impossible for JavaScript, as soon as a value is known to escape a function, its type becomes unknown. This los losss is simply because JavaScript is highly dynamic, and so predicting the effect of such flows is not generally possible. As such, type refinement tends to be quite precise as an intraprocedural technique, but less so as an interprocedural technique. In the presence of dynamic features such as reflection, code-loading, and polymorphism, useful analysis typically requires assuming some "well-behaved" subset of possible program behavior. Flow [5] takes this approach, assuming that certain dynamic features do not interfere with its analysis, at the cost of soundness. This work makes similar assumptions, but allows for ensuring soundness through optional run-time protection.

2.6 Related work

"Brand objects for nominal typing" by Jones et al [11] explores the introduction of types associated with a brand for gradual typing in a declaratively-typed setting. Declarative typing gives their system a superficial similarity to the one presented here, but is more restrictive in terms of when types can be declared. Because locus types are an application of type refinement, they can be applied to any object, including for instance mutations of existing objects with no prior type information, and to any method of object creation supported by the host language.

"The ins and outs of gradual type inference" by Rastogi et al [15] and "Principal type schemes for gradual programs" by Garcia and Cimini [8] also aim to reduce the syntactic burden of programmers in gradually-typed languages using type inference. However, their goals are mainly to determine the type of unannotated values based on their use, as in conventional type inference in languages like ML. Our goal is to make globally avaiable types that otherwise exist only locally. There is no reason that both systems could not be combined to further reduce the annotation burden on programmers.

"Refinement Types for TypeScript" by Vekris et al [24] suggests a system of refinement types in TypeScript, which again are orthogonal to type refinement. Note that while unfortunately similarly named, type refinement and refinement types are unrelated concepts: Refinement types are types with predicates, while type refinement is a procedure for determining precise types with imprecise or no types as input. The type refinement presented herein yields fairly conventional structural types, as the resulting types are simple and follow intuitive. Combining this naïvely with refinement types would likely not be useful, as any generated predicates would be as restrictive as the constructor, rather than general use of the object. Nothing in type system presented herein is incompatible with explicitly specified predicates, but explicit specification would be necessary.

Chapter 3

Motivation

In a setting such as JavaScript, the type of an object is not fixed by a type declaration; instead, type is a fluid notion which can change as an object is modified at run-time and its API evolves. Much of an object's API that would be fixed under a statically-typed language can be changed at run-time. A programmer's intuition of an object's type is often shaped by these dynamic changes. Indeed, this iterative mutation is the standard style of object creation in JavaScript, and so every notion of type the programmer has is locked to it.

In dynamic languages like JavaScript, even common idioms for object creation can confound traditional type systems. The conventional, widespread JavaScript practice is to create abstractions through fairly mechanical means:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.zeroDist = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
}
```

Traditional gradually-typed languages like TypeScript introduce types like traditional statically-typed languages: Through type declarations. In TypeScript, the `Point` class must be rewritten using declarative syntax to be understood by the type checker:

```
class Point {
    constructor(public x, public y) {}
    public zeroDist() {
        return Math.sqrt(this.x*this.x + this.y*this.y);
    }
```

```
}
```

This rewriting builds on the syntax of the new 6th version of the ECMAScript standard. While the latter construction may be typed easily, it is quite rigid, and is a significant syntactic change from convention. With locus types, imperative definitions can be analyzed as declarations. In LocusTypeScript, it is enough to add a single type annotation to introduce a globally usable `Point` locus type:

```
function Point(this: declare Point; x, y)
```

Using locus types, the assigned `x`, `y` fields along with the `zeroDist` method from `Point`'s prototype are automatically inferred. To have the coordinate fields typed with the desired `number` type one needs only to add local type annotations:

```
function Point(this: declare Point; x: number, y: number) {
    ...
}
Point.prototype.zeroDist = function() {...}
```

While TypeScript is able to provide a `Point` type with some refactoring, other common idioms are handled awkwardly at best. Consider a JavaScript programmer interfacing with a `Connection` API and wishing to add functionality for a game. The functionality consists of new `player` and `sendName` fields:

```
var conn = new Connection();
conn.player = player; // add player field dynamically
conn.sendName = function() { // add function field dynamically
    this.send(this.player.name); // assume 'player.name' exists
};
```

In a dynamically-typed language this behavior is easy to write and understand, but typing this behavior in a gradually-typed setting has traditionally been difficult. In TypeScript, providing an appropriate type to `conn` requires an unchecked downcast, and explicit specification.

Using locus types, with a single annotation, an augmented type can be extracted:

```
var conn: declare PlayerConnection = new Connection();
conn.player = player;
conn.sendName = function() {
    this.send(this.player.name);
};
```

To aid gradual typing, the inferred type is enforced during usage:

```
conn.sendName = "whoops"; // static error, incompatible type
```

In LocusTypeScript, it is dynamically checkable in a similar manner to JavaScript's '`instanceof`':

```
// checks for a tag associated with PlayerConnection:
if (conn declaredas PlayerConnection) {
    ... game-specific logic ...
}
```

Object creation in dynamic languages often follows patterns such as these. However, traditional static type systems such as that of TypeScript do not support such flow-sensitive forms of object creation. Because locus types are derived directly from analysis of imperative language constructs, they support this common style of object creation directly.

Chapter 4

Locus Types and LocusTypeScript

Conceptually, locus types are simply type annotations which reify the types inferred by type refinement. This concept is demonstrated by its implementation in LocusTypeScript. As a complete implementation with significant analogies to other languages, the properties of locus types are introduced by presenting the design of LocusTypeScript. As mentioned, LocusTypeScript is an an extension of StrongScript, which in turns extends TypeScript, which in turn extends JavaScript. While LocusTypeScript builds upon significant additions to the JavaScript language, its core purpose remains to show the utility of locus types in a JavaScript setting. Locus types, as a modular type system feature aimed at increased programmer convenience, require a conventional type system to be in place to truly be useful.

4.1 Overview

We intend to demonstrate the utility and applicability of locus types in a suitable gradually-typed language, through the design and implementation of LocusTypeScript.

From TypeScript and StrongScript, LocusTypeScript inherits the ability to annotate expressions with types: The special `any` type indicates that any type is acceptable. Expressions with the `any` type are essentially untyped. Types without a leading exclamation, e.g., `number`, represent optional types, and are checked statically if possible, and otherwise simply trusted. Types with a leading exclamation, e.g., `!number`, represent gradual types, and are checked statically if possible, and, failing that, at run-time. [1]

[1]Throughout this thesis the optionally-typed version of types is preferred in examples. The examples would trivially work with the concrete versions of types. The result would not change semantics apart from additional soundness checks (if needed).

To declare and use locus types, LocusTypeScript introduces two new annotations: `declare` and `becomes`. `declare` annotations act as locus type declarations, triggering analysis of the variable or parameter they annotate, as in the `Point` example. A `declare` annotation takes the form:

```
x : BaseType declare NewType
```

indicating that the object held by variable or parameter `x` begins as type `BaseType`, and, once the object is "complete", it is of type `NewType`. The structure of `NewType` is defined as the type which refinement infers for `x` at the end of its lexical scope. An object is considered complete at a code location if its inferred type at that location is a subtype of its type at the end of its scope:

```
function augment(x : BaseType declare NewType) {
    x.newField = 1; // derives 'newField : number'
    var xComplete : NewType = x; // 'x' must be complete
    x.newField = x.newField + 1; // valid, preserves structure
    // x.anotherNewField = 1; // invalid, changes structure
}
```

Assuming type refinement is correct, a complete object is therefore by definition compatible with its declared type. Typically, this occurs after the last field added to `x` is first assigned, as it must be a subtype of the final derived type.

The choice to install the new type at the end of a variable's scope has two surprising benefits: (1) It resolves the so-called "constructor problem", and (2) it supports multiple inheritance in a semantically-simple way.

The constructor problem, in short, is that an object must have a type during its own construction, but it cannot satisfy its own type until construction is complete. The type an object has mid-construction is the type derived by refinement up to that point, which is the lowest supertype of the `declare` type that it already satisfies.

Multiple inheritance is supported by flow-based type refinement unambiguously, in a manner similar to gaining fields:

```
var c: declare ColoredDrawableCircle = new Circle();
becomeColoredCircle(c); // gains 'ColoredCircle' base type
becomeDrawable(c); // gains 'Drawable' base type
```

Through the call to `becomeColoredCircle`, the relation `ColoredDrawableCircle` <: `ColoredCircle` is derived, and, similarly, through the call to `becomeDrawable`, the relation `ColoredDrawableCircle` <: `Drawable` is derived. By this mechanism, multiple inheritance is naturally supported, with predictable semantics.

The complementary `becomes` annotation takes the form:

```
x : BaseType becomes NewType
```

`becomes` annotations act as a means of expressing expected type transitions to the type system without introducing new types. Variables and parameters annotated with `becomes` are analyzed identically to those annotated with `declare`, but instead of giving a name to the inferred type, the inferred type is checked against the specified type. When used on a parameter, this allows functions to specify their type-transitioning behavior without declaring new types.

Because the annotations of parameters are part of a function's signature, `declare` and `becomes` declarations on parameters allow local type refinement information to be propagated across function boundaries.

For each `declare`, a locus type is defined, checkable by standard TypeScript/StrongScript rules. Flow-based type refinement is used to infer the structural properties of the annotated field or variable when it goes out of scope, and the inferred type becomes the structure of the declared locus type.

This concept can handle sophisticated dynamic cases. For instance, in the `Connection` example, the type of the connection object changes. The typical technique for handling this in a statically-typed language would be to make `PlayerConnection` a subclass of `Connection`, and override its constructor as appropriate. That technique is inflexible, as an object must gain its complete type during its initial construction. No such restriction exists in a dynamic language.

In our system, the type of `PlayerConnection` is the subtype of `Connection` with the additional fields `player` and `sendName` defined. All that was necessary to declare this type was to add an annotation to the variable representing `PlayerConnection`s during their initialization. Type refinement is sufficient to determine the structure of `PlayerConnection`, and because the base type of the declaring variable is `Connection`, `PlayerConnection` is a subtype of `Connection`.

Type refinement can be done lazily, as the full type needs to be known only when it is used in other code. As such, locus types can easily be recursive. Supporting recursive locus types without lazy evaluation is also possible.

Since type refinement provides a rich type for every variable at every code point, it is unnecessary for the variable to have its declared type within its scope. That is, the type specified to `declare` goes into effect when the value assigned to that variable goes out of scope, not locally.

4.2 Deriving Locus types

For the `PlayerConnection` example from chapter 3, assuming `player` and `sendName` do not already have incompatible types in the base type `Connection`, we trivially analyze the types of `player` and `sendName` as their last assigned type in the defining

scope. Conceptually, every field assignment causes a type refinement of `conn`, until the object has gained its final shape. Equivalently, we can describe these type-refining field assignments as the assignment plus a binding to a new variable of a more precise type, where all future references to the same variable are replaced with the further-refined version. Once the object gains its final shape, we use this shape to define `PlayerConnection`, as shown by the following pseudo-code representing locus type elaboration:

```
// Represents the steps towards locus type elaboration
// with pseudo-code type-refining assignments:
var conn0: Connection = new Connection();
conn0.player := player; // Add 'player' to type
// Re-assignment for clarity:
var conn1: Connection & {player: Player} = conn0;
conn1.sendName := function() {...} // Add 'sendName'
var conn2: typeof(conn1) & {sendName: () => void} = conn1;
// Finish locus type elaboration:
resolve PlayerConnection as typeof(conn2); // pseudo-code
```

Here `:=` represents a (hypothetical) type-refining assignment, with the type refinement made explicit through new declarations (for illustrative purposes). After intermediate type refinements, we resolve `PlayerConnection` to be equal to the final type of `conn2`, which is the input type of `conn`, `Connection`, with `player` and `sendName` fields added.

Object definition sites frequently have complications such as more than one possible exit path, causing conditional logic. Consider a directory type in LocusTypeScript defined with `declare` that gathers a file list from a directory, if possible.

```
function Dir(this: declare Dir; path) {
    var rd = readDirectory(path);
    if (rd === undefined) {
        this.good = false;      // : boolean
        return; // Early exit!
    }
    this.good = true;           // : boolean
    this.files = rd.getFiles();// : File[]
}
```

As with `Connection`, LocusTypeScript computes the members of `Dir` from its use in the lexical scope of its declaration. However, here we must take into account branches and exits in the code when analyzing `this`:

```
var this0: {} = this;
if (...) {
```

```
    this0.good := false;
    var this1: typeof(this0) & {good: boolean} = this0;
    resolve Dir as typeof(this1); // Exit 1, pseudo-code
    return; // Early exit!
}
// If branch may not have happened
var this2: typeof(this0) & {good: undefined|boolean} = this0;
this2.good := true;
var this3: typeof(this2) & {good: boolean} = this2;
this3.files := rd.getFiles();
var this4: typeof(this3) & {files: File[]} = this3;
// Finish locus type elaboration:
resolve Dir as typeof(this4); // Exit 2, pseudo-code
```

Here we have multiple `resolve` points for `PlayerConnection`, representing places where the variable goes out of scope. Once the algorithm has propagated types, LocusTypeScript computes the union of the types found at each scope exit. Thus, for `PlayerConnection`, the final type of `good` is analyzed as `boolean` $\cup$ `boolean` i.e., `boolean`, while `files` is analyzed as `undefined` $\cup$ `File[]`, i.e. `File[]`[2].

The analysis for LocusTypeScript, in constrast with that of languages such as Flow, does not attempt to reason semantically if `rd` could truly ever be `undefined`. Furthermore, when typing `this2`, the `good` field is considered to possibly be assigned, even though language semantics make this impossible. Since locus types are globally usable, a simple scheme for handling control flow aids programmer reasoning and prevents brittleness. The details of this derivation process are shown in Section 6.

4.3 Handling interprocedural flow

When analyzing the `Connection` example, it is enough to consider the starting type and subsequent assignments to compute the final type. Consider, however, if the `Connection` example is defined through the use of a helper function instead of direct assignments:

```
function addPlayerMethods(conn) {
    conn.player = player;
    conn.sendName = function() {
        this.send(this.player.name);
    };
```

[2]Array types in LocusTypeScript are non-concrete, and non-concrete types are supertypes of `undefined`.

```
}
var conn: declare PlayerConnection = new Connection();
addPlayerMethods(conn);
```

The previous local analysis does not suffice. While a `PlayerConnection` type is still computed, it is no longer able to analyze the assignment of the `player` and `sendName` types. Some languages such as Flow eagerly investigate the effects of functions such as `addPlayerMethods`. To avoid unexpected dependencies that can hinder modularity, however, LocusTypeScript requires the programmer to opt-in to interprocedural analysis. LocusTypeScript provides two semantically equivalent ways to achieve this. Firstly, the definition of `PlayerConnection` can be moved to the `addPlayerMethods` parameter:

```
function addPlayerMethods(
        conn: Connection declare PlayerConnection) {
    conn.player = player;
    conn.sendName = function() {
        this.send(this.player.name);
    };
}
var c = new Connection();
addPlayerMethods(c);
```

By moving the `declare` annotation to the `conn` parameter, we are able to include the results of flow-based analysis from that function. After the call to `addPlayerMethods`, `conn` becomes an object of type `PlayerConnection`.

However, moving the declaration site is not always possible. For example, after calling `addPlayerMethods`, we might perform additional changes. To support this, we add a simple facility to LocusTypeScript to inherit changes undergone by a function parameter:

```
function addPlayerMethods(conn: Connection declare) {
    conn.player = player;
    conn.sendName = function() {
        this.send(this.player.name);
    };
}
var c: declare PlayerConnection = new Connection();
addPlayerMethods(c);
// Example additional change, an additional field in
// PlayerConnection:
c.gameState = getGameState();
```

Here we use a `declare` without a type name, avoiding namespace pollution but still propagating an intermediate type. The fact that `c` is extended to include the new, anonymous type is part of the signature of `addPlayerMethods`. `PlayerConnection` is known to be a subtype of the anonymous type defined by `addPlayerMethods`, through analysis of the `addPlayerMethods` call.

While `declare` allows for type modifications to propagate across procedures, it introduces essentially dynamic semantics on the variable, which is not always desired. To that end, the similar `becomes` annotation allows for specifying exactly the changes that are expected:

```
// Defines the 'PlayerConnection' type:
function addPlayerMethods(
        conn: Connection declare PlayerConnection) {
    conn.player = player;
    conn.sendName = function() {
        this.send(this.player.name);
    };
}
// Uses the 'PlayerConnection' type, ensuring it propagates:
function handshake(
        conn: Connection becomes PlayerConnection) {
    addPlayerMethods(conn);
    conn.sendName();
}
var c = new Connection();
handshake(c); // 'c' gains 'PlayerConnection'
```

During the analysis of `c`, it is assumed that the `becomes` specification is true (i.e., `conn` does become a `PlayerConnection`). This `becomes` type is used as a requirement when analyzing `handshake`. If `conn` does not indeed become a `PlayerConnection`, the analysis fails and the program does not type check. This is in contrast to `declare`, which allows arbitrary type expansions.

To type check the `becomes` specification, we analyze `conn` in `handshake`. Since `conn` is known to pass through a parameter that gains the `PlayerConnection` type, we can infer that `conn` is of type `PlayerConnection` when the function ends, and the `becomes` specification type checks.

4.4 Type granularity

Type refinement generalizes to data-flow analysis. In theory, locus types could be as specific as individual values or ranges. For instance, if `false` is assigned to a field,

the field could, in theory, be given the type `false` instead of `boolean`. In practice, it is unlikely to capture the intent of code. There is no "perfect" type, so when determining types of object members, we generalize by following the type inference rules already present in TypeScript for assignments. It is presumed that these mirror programmer intent.

4.5 Soundness

`declare`-typed references have some of the dynamic properties of dynamically-typed (`any`-typed) references, in that they allow fields not specified in the type to be written, extending the object. These references can have a base type, in which case assignments that conflict with the base type are not allowed, but this still allows a source of unsoundness: If the declared base type is not the most specific correct type for the object, then the declaration may attempt to add a field that conflicts with an existing field not specified in the type. This is a natural consequence of objects gaining types through mutation. Other aliases to the same value with more specific types may find that their types are unexpectedly violated, and changes made through aliases with less specific types will not be reflected into the locus type. Nonetheless, if `declare` is only used for initial construction of objects, equivalently to conventional static types, it is trivially sound: The value cannot have a more specific type than `Object`, and its extension cannot fail. Using `declare` to extend objects in other circumstances exposes this unsoundness, but is a power not offered by other optionally or gradually-typed systems, and there is likely no sound equivalent. It is thereby contended that `declare` behaves like a principled `any`, and likely reflects exactly the unsoundness that a user of an optionally or gradually-typed language would expect it to.

In the gradual typing community, substantial work has been done on assigning blame when such violations occur from dynamic code[26], and `declare` code is no different. In the setting of a checked, gradually-typed language, these unsound field additions would fail, either due to a contract violation or, in the case of LocusTypeScript, due to the member type protection inherited from StrongScript, as discussed in Section 4.7.

Finally, our IDE tools provide a path from locus types to conventional, TypeScript types, which makes explicit where dynamic behavior occurs.

4.6 Nominality and checkability

LocusTypeScript takes the design choice of making locus types nominal and runtime checkable through brands like in Modula-3 [14]. Conceptually, brands take a

structural type system and make it a nominal type system through fields (termed brands) unique to a certain type. This branding causes minor run-time overhead, with the benefit that locus type membership can easily be checked. This checkability aids the task of gradual typing. For example, when calling a function of unsure origin with a "`becomes`" or "`declare`" annotation, it must be checked that the target type is actually achieved at run-time. The trivial checkability of locus types through their installed brands helps in this case. However, checkability is not fundamental to locus types. Locus types work equality well as structural types without additional brand members. This design is a good fit for an optionally-typed language such as TypeScript (without StrongScript extensions).

4.7 Interaction with gradual types

There is a range of design decisions in a gradually-typed programming language, and those decisions affect locus types. We chose to implement LocusTypeScript on StrongScript, as it provided an environment for sound, gradual typing on top of JavaScript.

In the absence of StrongScript's concrete types, LocusTypeScript has the same dynamic semantics as JavaScript, and only adds static type checking. The only new run-time operation is the installation of a checkable brand on the object, indicating that it is a complete object of the analyzed type. Since each brand associated with a type is unique, this installation never fails[3]. Thus, locus type reification itself is trace preserving (i.e., does not alter semantics) [17].

StrongScript's types are always checkable eagerly because its type system is nominal. Type checks are fast and nonrecursive. Locus types complement nominal types well: The name corresponds precisely to those objects that have reached the locus type declaration. With run-time enforcement, branded objects have the same type correctness as objects of other nominal types. Locus types can thereby be treated similarly to brands [14] by simply adding a marker to the relevant objects. Code is inserted wherever locus types are declared to mark objects with a "brand" unique to that type. This branding is done invisibly[4] in a hidden object field, and allows all locus types to have an eagerly checkable concrete type.

Another obvious choice in the design spectrum is unsound, structural types. This option creates types as in TypeScript, and would be a reasonable approach for that

[3]In ECMAScript, objects can be "frozen", preventing all updates, including branding. A frozen object is not brandable, but since it cannot be modified, it would probably have failed to assume the type long before it failed to be branded.

[4]To the degree allowed by JavaScript.

language. All that is necessary is to not brand or perform run-time checks.

Other gradually-typed languages [22][25] use sound structural typing. Locus types infer a structural type, so all that is necessary to use them in such a language is to not implement branding. The resulting inferred types have all the same benefits and drawbacks as existing structural types in these systems.

4.8 Run-time protection

While not fundamental to the concept of locus types, some design decisions aiding gradual typing have an impact on run-time behavior. Inheriting from StrongScript, run-time protection is applied to certain language constructs, such as concrete type annotations, and function or method members of modules and classes. This protection allows for stronger reasoning guarantees about these annotations and functions, used to guarantee type constraints are sound. While this may cause some run-time incompatibility, for example by disallowing dynamic replacement of module functions, immutability generally is intended when using these construct. These features of StrongScript are largely unchanged, and discussed in detail in [17].

Additionally, LocusTypeScript uses run-time protection at locus type declaration sites. When used with sound types, such as StrongScript's concrete types, the analyzed structure must be protected at run-time to assure that inferred type information remains valid. In LocusTypeScript, this is implemented via JavaScript accessors. Through this mechanism, flow-based type refinement can be made sound in the face of uncertainty due to dynamic features. Most gradually-typed programming languages have some system of contracts, wrappers or accessors to assure this. Locus types need only to use the same system.

Runtime protection is only applied to members that evaluate to a concrete type, or a union of concrete types. Consider the following (sanitized for readability) run-time output produced from the `Connection` example:

```
// In module scope:
var PlayerConnection = new LocusType();
// In defining scope:
var conn = new Connection();
// Restrict 'player' re-assignment:
add(conn, "player", player, Player);
// Install unwrapped method:
addHidden(conn,"raw_sendName",function(){
  this.raw_send(this.raw_player.raw_name);
});
```

```
// Install proxy method:
addFunc(conn, "sendName", function() {
    check(this, PlayerConnection); // perform type check
    this.raw_sendName();
});
addLocusType(conn, PlayerConnection); // brand once complete
```

In the outer scope of the LocusTypeScript module, a `PlayerConnection` locus type object is created, used to mark objects as belonging to that locus type. `addLocusType` then installs this type on a cached immutable list in `conn`, dynamically enforceable via the `check` function.

Note that the `raw_*` fields and functions are name-mangled in actual implementation to indicate they are not intended for use by the programmer. Due to the nature of JavaScript, however, defeating the type system by accessing them directly is always possible [17]. ECMAScript 6 symbols can be used to create truly private members if certain run-time facilities are overridden. However, as ECMAScript 5 is currently the most widely used version of JavaScript, backwards compatibility was desired.

In the run-time protection illustrated above, the `addHidden` function operates like a normal assignment, except configuring the JavaScript property to be unenumerable if not already configured (hidden from object reflection).

The `add` run-time function acts like a special assignment operator which installs validated accessor functions (getter/setter pair) for untyped code.

The `addFunc` function is stricter. While allowing access from untyped code, all attempts to set the field that do not stem from properly typed function literals will result in errors. This strictness could be loosened somewhat if function types were run-time checkable in the underlying system.

All of the `add` functions utilize JavaScript's `Object.defineProperty`. If an existing property created with `Object.defineProperty` is detected, object creation fails and an exception is thrown.

In StrongScript, fields and parameters with "concrete" annotations protect their type declarations by intercepting accesses from untyped code and eagerly checking the types. Concrete types, written with the `!` symbol in StrongScript, are subtypes of their non-concrete equivalent. For instance, `!Connection <: Connection`. Without the concreteness annotation, types are unsound by design. Locus type inference is orthogonal to concreteness; values that are specified or inferred as concrete derive concrete types. This does imply that adding a `declare` can have semantic effects: Since typed objects protect themselves, incorrect access raises errors, equivalently to having manually specified the corresponding type for the field. Assuming protections

can be installed[5], semantic differences other than raising such type errors do not occur.

4.9 Backwards compatiblity

Locus types are first and foremost an alternative way of specifying types. It is possible, however, that they allow for certain values to be typed in a way that the host language cannot specify, due to their dynamic nature. For instance, in LocusTypeScript, it is possible for an object to gain a new nominal type at run-time, which is impossible in StrongScript. Used for more traditional types, in particular during object construction, they can always be expressed equivalently with some explicit type. In languages with wrappers for sound structural typing, and of course in unsound languages, every locus type could be expressed equivalently with some explicit type.

Since the underlying type system is unmodified, there is no consequence to mixing locus types with other gradual types. Theoretically, a system could have any mix of dynamic, locus and traditional static types. We expect that the usual evolution of any given type would be from purely dynamic, to locus, to traditional static.

LocusTypeScript aims to remain run-time compatible to JavaScript to the extent possible. Run-time protection means that adding a `declare` annotation can affect the semantics of a program. However, the same changes would occur had the code been rewritten with explicit types in a gradually-typed language. This protection is a common trade-off when soundness is desired, and is encountered in existing gradually-typed programming languages. Dynamic type guarantees for LocusTypeScript are achieved through run-time checks and protection on fields through the use of JavaScript accessors. If existing accessors are present on that field not known to LocusTypeScript, then a run-time error occurs. Objects correctly inherit these custom fields through their prototype objects.

LocusTypeScript protects the most general derived type it computes for a member. As a result, some usage patterns require type annotations to ensure correct type inference, such as when assigning an instance of a derived type, when the base type was intended:

```
var obj: declare HasBase = {};
// Must annotate to preserve intent:
obj.base = <Base> new Derived();
```

Locus type objects are imported to other modules when the locus type names are imported, allowing for modular programming. In LocusTypeScript, each object

[5]Accessor installation can fail if existing accessors have already been installed.

maintains a cached list of locus types they belong to. This side information does not interfere with the existing prototype object hierarchy. Immutability ensures the list is properly shared between a prototype object and its extenders.

Since function objects do not have associated signature information, reassigning bound function expressions is allowed only in typed code. Accomodating this restriction typically only requires minor adjustment, such as adding additional type declarations. If this protection is too strict, an indirect assignment can be used. Consider a modification of the `PlayerConnection` example, designed to sidestep this protection:

```
var conn: declare PlayerConnection = new Connection();
conn.player = player;
var sendName = function() {
    this.send(this.player.name);
};
conn.sendName = sendName;
```

Unlike in the original example, the function is first assigned to a temporary variable. As a result, no run-time protection occurs, since function objects cease to be concretely typed once they are assigned. This level of protection was chosen as we anticipate that method replacement is rare, and so being conservative in their protection is reasonable.

In general, if an error is raised and exits the defining scope, locus type objects such as `conn` are left in an incomplete state. However, either the object is completed or the `addLocusType` call is not reached. Without the locus type attached to the object, it fails to coerce to a locus type at run-time, preventing misuse.

The installation of a locus type is semantically similar to the construction of a class in StrongScript. Because the branding occurs at the end, it additionally avoids the constructor problem. The StrongScript paper [17] includes proofs that the type system of LocusTypeScript without locus types is sound.

Concrete/sound type annotations are checked at run-time. Function members in modules and classes are frozen to enable optimizations that avoid dynamic checks. Runtime protection facilities are inherited from StrongScript.

4.10 Tooling

To aid programming with locus types, a modified version of the `atom-typescript` plugin was developed using TypeScript's tooling API. This plugin brings traditional IDE features for LocusTypeScript to the Atom text editor. On-the-fly type checking

is provided, allowing locus type sites to be changed with automatic feedback provided if existing types are broken. Type introspection and field autocompletion allow programmers to scrutinize locus types, mitigating disadvantages their implicit nature may have.

Standard refactoring options such as variable renaming are inherited from TypeScript, as well as features specifically for working with locus types. As the inferred type is available to the IDE, a refactoring option exists to make the inferred type explicit. This feature allows programmers to use the inferred structure as a quick starting point. Even a programmer uninterested in the benefits of locus types could benefit from the concept by writing syntactically-light locus types and refactoring them automatically into explicit types.

It is additionally possible that a static or dynamic analysis tool could automatically find sites likely to be viable for locus types by using heuristics. With such a tool, a programmer could quickly move from wholly untyped code to code with at least some useful types.

Chapter 5

Formal Properties

Locus types are a merger of two orthogonal technologies, namely gradual typing and type refinement, and for brevity we do not reiterate the formal properties of both. Instead, we focus on presenting the issues specific to implementing locus types. In contrast to a typical JavaScript semantics, we require the ability to represent type expanding objects, and we introduce primitives for protected fields and a form of object tagging that upholds type invariants for locus type brands. We assume that a set of locus type brands L has been provided, and a function that gives the additional object fields/brands, which necessarily accompany it, denoted `elaborate`(L). `elaborate` essentially encompasses the inference necessary to discover locus types, while we present here a core semantics, assuming `elaborate` has been resolved, that allows for expanding objects in a manner the type system is aware of. `elaborate` is resolved at a higher level, through flow-based type refinement discussed in Section 4.3.

5.1 Core semantics

We formalize the relevant core language of LocusTypeScript in a similar fashion as [17]. We present relevant deviations from [17], which itself builds upon the functional core of λ_{JS}. We formalize LocusTypeScript as an extension to λ_{JS} of [9]; a distilled formal semantics for JavaScript. We equip λ_{JS} with a structural type system, describing object types that contain protected type fields and locus type brands. The interaction between concrete and non-concrete types remains largely inherited from StrongScript, and is formalized in [17]. Unlike [17], we formalize only the interaction between the `any` dynamic type and concrete types. The relevant typing and evaluation rules are given in Figure 5.1, which formalizes the core typesystem of LocusTypeScript. The structural type system is inherited from TypeScript and

formalized in [1].

Syntax. A program consists of a collection of locus types plus an expression to be evaluated. The type of dynamic expressions is `any`. `C` ranges over core classes, and classifies core language types such as `null`, `undefined`, as well as number, boolean, and string objects. In addition to unprotected fields (regular JavaScript fields), objects can have protected fields that protect writes to an object, ensuring that the field is always a subtype of some type. Unlike StrongScript, union types feature in the formalization, as they occur naturally when deriving locus types. s represents the type of string literals, $t_1\,..\,\rightarrow t$ denotes function types, and the object type $\{s_1{:}t_1\,..\mid L_1..\}$ denotes objects with protected fields $s_1{:}t_1\,..$ and locus type brands $L_1..$ [1]:

$$t \;::=\; \texttt{C} \mid \texttt{any} \mid t_1\,..\,\rightarrow t \mid \{\; s_1{:}t_1\,..\mid L_1..\;\} \mid t_1 \cup t_2$$

As object types deal with rigidly protected fields, an object subtype must contain the exact field types of the supertype, as stated by SObj (Figure 5.1):

$$\{s_1{:}t_1..\; s_m{:}t_m..\mid L_1..\; L_2..\} <: \{s_1{:}t_1..\mid L_1..\}$$

This implies that a more specific protected field cannot act as a less specific protected field, and vice versa. However, the actual value held by the field can be of a subtype. We assume the presence of two conversion functions, `toString` and `toBoolean`, which coerce values into the relevant JavaScript primitives. We omit the definition of these standard JavaScript operations.

We omit the definition of a `type` function for values, which categorizes values into their corresponding types. This function essentially applies rules TObj and TFunc (defined in Section 5.1).

Expressions are inherited from λ_{JS} with some modifications:

$$\begin{aligned} e \;::=\;& x\ [\textsc{Var}] \mid \{\; (s_1{:}e_1 \mid t_1)\,..\mid L_1..\;\}\ [\textsc{Object}] \\ \mid\;& e_1[e_2]\ [\textsc{Get}] \mid e_{1[t]}[e_2] = e_3\ [\textsc{Update}] \\ \mid\;& \langle t\rangle e\ [\textsc{Cast}] \mid e\ \texttt{declare}\ L\ [\textsc{AddBrand}] \\ \mid\;& \texttt{let}\ (x{:}t = e_1)\ e_2\ [\textsc{Let}] \mid \texttt{delete}\ e_1[e_2]\ [\textsc{Delete}] \\ \mid\;& \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\}\ [\textsc{Func}] \mid e(e_1..)\ [\textsc{App}] \\ \mid\;& \texttt{if}\ (e)\ \{e_{\text{l}}\}\ \texttt{else}\ \{e_{\text{r}}\}\ [\textsc{If}] \end{aligned}$$

[1] Throughout, we denote lists l_1 through l_n simply as "l_1..". This list may be empty. If there is only one list of its type and its contents are not relevant, we abbreviate it further as simply "..". We denote the concatenation of two lists as "$l_1..\ l_m$..". We denote an element appended to the list as "$l_1..l$". We assume lists to have set-like semantics, such that appending duplicate elements is a no-op.

Functions and let bindings are explicitly typed, and expressions can be cast to arbitrary types (with run-time validation). Objects, denoted $\{(s_1{:}e_1|t_1)\,..\mid L_1..\}$, in addition to the fields' values, carry type tags for protected fields, and locus type brands. Field tags allow for enforcing type soundness when objects are used dynamically, with `any` representing unprotected fields. Typical dynamic JavaScript objects have fields only tagged with `any`, and no locus type brands.

We assume the presence of a `type` function for values, which categorizes values into the appropriate type, by categorizing primitives and applying rules such as TObj and TFunc.

Evaluation contexts are defined as follows:

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \texttt{let}\,(x{:}t = E)e_2 \mid E_{\langle t\rangle}[e] \mid v_{\langle t\rangle}[E] \mid E[e_2] = e_3 \mid v[E] = e_3 \\
& \mid & v_1[v_2] = E \mid E(e_1\,..\,e_n) \mid v(v_1\,..\,v_n, E, e_1\,..\,e_k) \mid E\ \texttt{declare}\ L \\
& \mid & \{(s_1{:}v_1 \mid t_1)..\ (s{:}E \mid t)\ (s_m{:}e_m \mid t_m)..\mid L_1..\} \mid \texttt{delete}\,E[e] \mid \texttt{delete}\,v[E] \\
& \mid & \langle t\rangle E \mid \texttt{if}\ (E)\ \{e_2\}\ \texttt{else}\ \{e_3\} \mid \texttt{if}\ (v_1)\ \{E\}\ \texttt{else}\ \{e_3\} \\
& \mid & \texttt{if}\ (v_1)\ \{v_2\}\ \texttt{else}\ \{E\}
\end{array}
$$

$$\frac{t <: t' \quad t'_1 <: t_1 ..}{t_1.. \to t <: t'_1.. \to t'} \text{[SFunc]}$$

$$\frac{t_1 <: t_3 \quad t_2 <: t_3}{t_1 \cup t_2 <: t_3} \text{[SUnionTo]}$$

$$\frac{t_3 <: t_2 \vee t_3 <: t_1}{t_3 <: t_1 \cup t_2} \text{[SUnionFrom]}$$

$$\frac{}{\{s_1{:}t_1..\ s_m{:}t_m.. \mid L_1..\ L_2..\} <: \{s_1{:}t_1..|L_1..\}} \text{[SObject]}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \text{[TVar]}$$

$$\frac{\Gamma \vdash e : t_1 \quad t_1 <: t_2}{\Gamma \vdash e : t_2} \text{[TSub]}$$

$$\frac{x_1{:}t_1.., \Gamma \vdash e : t}{\Gamma \vdash \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\} : t_1.. \to t} \text{[TFunc]}$$

$$\frac{\Gamma \vdash e : \mathsf{any}}{\Gamma \vdash e_1 : t_1 ..} \quad \frac{}{\Gamma \vdash e(e_1..) : \mathsf{any}} \text{[TAppAny]}$$

$$\frac{\Gamma \vdash e : t_1.. \to t \quad \Gamma \vdash e_1 : t_1 ..}{\Gamma \vdash e(e_1..) : t} \text{[TApp]}$$

$$\frac{\Gamma \vdash e_1 : t \quad x{:}t, \Gamma \vdash e_2 : t'}{\Gamma \vdash \texttt{let}\ (x{:}t = e_1)\ e_2 : t'} \text{[TLet]}$$

$$\frac{\Gamma \vdash e : \{s_1{:}t_1..|..\} \quad \{s_1{:}t_1..|..\} <: \mathsf{elaborate}(L)}{\Gamma \vdash e\ \texttt{declare}\ L : \{s_1{:}t_1..|..\ L\}} \text{[TAddBrand]}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1[e_2] : \mathsf{any}} \text{[TGetDynamic]}$$

$$\frac{\Gamma \vdash e_1 : \{s_1 : t_1..\ s : t_r \mid ..\}}{\Gamma \vdash e_1[s] : t_r} \text{[TGet]}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{delete}\ e_1[e_2] : \mathsf{any}} \text{[TDelete]}$$

$$\frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e_l : t_2 \quad \Gamma \vdash e_r : t_2}{\Gamma \vdash \texttt{if}\ (e)\ \{e_l\}\ \texttt{else}\ \{e_r\} : t_2} \text{[TIf]}$$

$$\frac{\Gamma \vdash e : t_1 \quad t_1 = \mathsf{any} \vee t_2 = \mathsf{any} \vee t_1 <: t_2 \vee t_2 <: t_1}{\Gamma \vdash \langle t_2 \rangle e : t_2} \text{[TCast]}$$

$$\frac{\Gamma \vdash e_l{:}t_l \quad \Gamma \vdash e_s{:}t_s \quad \Gamma \vdash e_r{:}t_r \quad \texttt{is\_not\_string\_literal}(e_s)}{\Gamma \vdash e_{l[\mathsf{any}]}[e_s] = e_r : t_l} \text{[TUpdateDynamic]}$$

$$\frac{\Gamma \vdash e_1 : \{s_1{:}t_1..\ s{:}t_p|..\} \quad \Gamma \vdash e_r : t_r \quad t_p = \mathsf{any} \vee t_r <: t_p}{\Gamma \vdash e_{l[t_p]}[s] = e_r : \{s_1{:}t_1..\ s{:}t_r|..\}} \text{[TUpdate]}$$

$$\frac{\Gamma \vdash e_l : \{s_1{:}t_1..|..\} \quad \Gamma \vdash e_r : t_p \quad s \notin \{s_1..\}}{\Gamma \vdash e_{l[t_p]}[s] = e_r : \{s_1{:}t_1..\ s{:}t_p|..\}} \text{[TUpdateCreate]}$$

$$\frac{\forall L_i \in \{L_1..\}.\ \{s_1{:}t_1..|L_1..\} <: \mathsf{elaborate}(L_i)}{\Gamma \vdash \{(s_1{:}v_1|t_1)..|L_1..\} : \{s_1{:}t_1..|L_1..\}} \text{[TObj]}$$

Figure 5.1: The type system

$$\frac{s \notin \{s_1..\} \quad \texttt{"__proto__"} \notin \{s_1..\}}{\{(s_1{:}v_1|t_1)..\mid ..\}[s] \longrightarrow \texttt{undefined}}\ [\textsc{EGetNotFound}]$$

$$\frac{}{(\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\})(v_1..) \longrightarrow e\{v_1/x_1..\}}\ [\textsc{EApp}]$$

$$\frac{}{\{..\ (s{:}v|t)..\mid ..\}[s] \longrightarrow v}\ [\textsc{EGet}]$$

$$\frac{\neg\texttt{is\_string}(v)}{\{..\}[v] \longrightarrow \{..\}[\texttt{toString}(v)]}\ [\textsc{EGetToString}]$$

$$\frac{\mathsf{is\_not\_func}(v) \quad \texttt{type}(v) <: t}{\langle t\rangle v \longrightarrow v}\ [\textsc{ECast}]$$

$$\frac{s \notin \{s_1..\}}{\{(s_1{:}v_1|t_1)..\ (\texttt{"__proto__"}{:}v|t)..\mid ..\}[s] \longrightarrow v[s]}\ [\textsc{EGetProto}]$$

$$\frac{t' = t'_1.. \to t'' \vee (t' = \mathsf{any} \wedge t'_1 = \mathsf{any}\ ..\ \wedge t'' = \mathsf{any})}{\langle t'\rangle(\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\}) \longrightarrow \texttt{func}(x_1{:}t'_1..)\{\texttt{return}\ \langle t''\rangle((\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t'\})(\langle t_1\rangle x_1)..) : t''\}}\ [\textsc{EFun}]$$

$$\frac{(t_p = \mathsf{any} \wedge t = t') \vee (t_p = t = t') \vee (t_p = t' \wedge t = \mathsf{any})}{\{..\ (s{:}v|t)..\mid ..\}_{[t_p]}[s] = v' \longrightarrow \{..\ (s{:}\langle t'\rangle v'|t')..\mid ..\}}\ [\textsc{EUpdate}]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ [\textsc{ECtx}] \qquad \frac{s \notin \{s_1..\}}{\{(s_1{:}v_1|t_1)..\mid ..\}_{[t]}[s] = v' \longrightarrow \{(s_1{:}v_1|t_1)..\ (s{:}v'|t)\mid ..\}}\ [\textsc{EUpdateNotFound}]$$

$$\frac{\texttt{toBoolean}(v) = \texttt{true}}{\texttt{if}\ (v)\ \{e_l\}\ \texttt{else}\ \{e_r\} \longrightarrow e_l}\ [\textsc{EIfTrue}] \qquad \frac{\texttt{toBoolean}(v) = \texttt{false}}{\texttt{if}\ (v)\ \{e_l\}\ \texttt{else}\ \{e_r\} \longrightarrow e_r}\ [\textsc{EIfFalse}]$$

$$\frac{}{\texttt{delete}\,\{..\ (s{:}v|\mathsf{any})..\mid ..\}[s] \longrightarrow \{\ ..\mid ..\}}\ [\textsc{EDelete}] \qquad \frac{\neg\texttt{is\_string}(v)}{\{..\}_{[t_p]}[v] = v' \longrightarrow \{..\}[\texttt{toString}(v)] = v'}\ [\textsc{EUpdateToString}]$$

$$\frac{\neg\texttt{is\_string}(v_2)}{\texttt{delete}\,v_1[v_2] \longrightarrow \texttt{delete}\,v_1[\texttt{toString}(v_2)]}\ [\textsc{EDeleteToString}] \qquad \frac{}{\{\ ..\mid ..\}\ \texttt{declare}\ L \longrightarrow \{\ ..\mid ..\ L\}}\ [\textsc{EAddBrand}]$$

$$\frac{s \notin \{s_1..\}}{\texttt{delete}\,\{(s_1{:}v_1|t_1)..\mid ..\}[s] \longrightarrow \{(s_1{:}v_1|t_1)..\mid ..\}}\ [\textsc{EDeleteNotFound}] \qquad \frac{}{\texttt{let}\ (x{:}t = v)\ e \longrightarrow e\{v/x\}}\ [\textsc{ELet}]$$

Figure 5.2: The run-time semantics

5.2 Metatheory

In LocusTypeScript, *values* are expressions that have evaluated to functions, and objects whose fields contain values. We say that an expression is *stuck* if it is not a value and no reduction rule applies. We say that an expression is *well-typed* if a type can be given to it through the typing rules (intuitively representing a program without type errors).

Lemma 5.2.1 (Step-wise Preservation)**.** *Given a well-typed expression* $\Gamma \vdash e_{before} : t_{before}$*, if* $e_{before} \longrightarrow e_{after}$*, then* $\Gamma \vdash e_{after} : t_{before}$*.*

Proof: The lemma is considered for every possible construction of e_{before}.

If e_{before} is a VAR or FUNC then the expression is a value and the lemma is vacuously true (since no further evaluation occurs).

If e_{before} is an OBJECT, and presuming the lemma holds for its component expressions $e_1..$, then ECTXT preserves the object type t_{before}, and $e_{after} : t_{before}$.

If e_{before} is an IF, then both sub-expressions e_l and e_r are typed as t_{before}. Regardless of which is used when evaluating EIF, $e_{after} : t_{before}$.

If e_{before} is an APP, then EAPP substitutes $x_1..$ for $v_1..$, where $x_1..$ and $v_1..$ match in type, and evaluates e with this substitution. Since $e : t_{before}$, and the substitution $e\{v_1/x_1..\}$ does not change this type, then $e_{after} : t_{before}$.

If e_{before} is a LET and typed as t_{before}, this implies the evaluated expression e_2 is also typed as t_{before}. Since $e_2 : t_{before}$, and the substitution $e\{v/x\}$ does not affect this type, then $e_{after} : t_{before}$.

If e_{before} is an ADDBRAND then $t_{before} = \{.. \mid .. L\}$, modifying the type of e to add a type tag L if it is not already present. Since this mirrors the addition of L during EADDBRAND, $e_{after} : t_{before}$.

If e_{before} is a DELETE then $e_{after} : t_{before} = \texttt{any}$ from the definition of TDELETE.

If e_{before} is an UPDATE where TUPDATEDYNAMIC applies, then either evaluation gets stuck and the lemma is vacuously true, or a field is updated with an appropriate type. Since the object type is preserved, $e_{after} : t_{before}$.

If e_{before} is an UPDATE where TUPDATE applies, then either evaluation gets stuck and the lemma is vacuously true, or a field without protection is updated/created. Since fields without protection do not affect the objects type, $e_{after} : t_{before}$, then $t_{before} = \texttt{any}$, which can apply to any expression, thus $e_{after} : t_{before}$.

If e_{before} is a CAST, and the casting type (t_{before}) is a non-function type, evaluation gets stuck during ECAST if e is not typable as t_{before}. If e_{before} is a CAST, and t_{before} is a function type, then the value is wrapped in a function of the appropriate type. Therefore, in both cases, after the cast evaluates then $e_{after} : t_{before}$.

If e_{before} is a GET where TGETDYNAMIC applies and does not get stuck then trivially $e_{after} : t_{before} = \texttt{any}$. If e_{before} is a GET where TGET applies. Since e_{before} is defined as well-typed, then e_1 has a component s of type $t_r = t_{before}$. Through rule EGET, we extract this component as e_{after}, and thus $e_{after} : t_{before}$.

The lemma is true for every possible construction of e_{before}, and thus the lemma holds. ■

Theorem 5.2.2 (Preservation)**.** *Given a well-typed expression $\Gamma \vdash e : t$, if $e \longrightarrow^* v$, then $\Gamma \vdash v : t$.*

Proof: Preservation follows from the Step-wise Preservation lemma. Since the Step-wise Preservation lemma ensures that types are preserved for any individual step, they must also be preserved for an arbitrary sequence of steps. Note that Step-wise Preservation lemma requires each intermediate step to be well-typed; this condition is preserved along with typings. ■

Lemma 5.2.3 (Step-wise Progress)**.** *A "strictly-typed" expression $\Gamma \vdash e : t$ is defined as a well-typed expression with each sub-expression e_{sub} satisfying $e_{sub} : t_{sub} \neq \texttt{any}$, each* CAST *sub-expression $\langle t_{cast} \rangle e_{sub}$ satisfying $e_{sub} : t_{sub} <: t_{cast}$ (i.e, no down-casts), and each* UPDATE *sub-expression having the form $e_{l[t_p]}[s] = e_r$ satisfying $t_p = t_{field}(s) \neq \texttt{any}$, where t_{field} is a function relating each label s to a consistently used type $t_{field}(s)$, and each* OBJECT *sub-expression having its labels $s_1 .. : t_{field}(s_1) ..$.*

Given a "strictly-typed" expression $\Gamma \vdash e_{before} : t_{before}$, either $e_{before} \longrightarrow e_{after}$ for a "strictly-typed" e_{after}, or e_{before} is a value.

Proof: We show, through deconstruction of every case, that either $e_{before} \longrightarrow e_{after}$ or e_{before} is a value. It is implied that e_{after} is "strictly-typed" due to lack of introduction of sub-expressions that would break the conditions presented, unless otherwise noted (namely, for CAST and UPDATE).

If e_{before} is a VAR or FUNC then e_{before} is a value.

If e_{before} is an OBJECT, then $e_{before} \longrightarrow e_{after}$ through ECTXT unless e_{before} is a value. Therefore, either $e_{before} \longrightarrow e_{after}$ or e_{before} is a value:

If e_{before} is an IF, then $e_{before} \longrightarrow e_{after}$ through EIF.

If e_{before} is an APP, then the typing rule TAPP is used (since $e : t_{sub} \neq \texttt{any}$), then EAPP always applies, as e must evaluate to a function value of the correct arity, then $e_{before} \longrightarrow e_{after}$ through EAPP.

If e_{before} is a LET, then $e_{before} \longrightarrow e_{after}$ through ELET.

If e_{before} is an ADDBRAND then EADDBRAND applies as $e : \{ .. \mid .. \}$ (by TADDBRAND), and thus $e_{before} \longrightarrow e_{after}$ through EADDBRAND, which applies to any value with an object type.

If e_{before} is a DELETE, then TDELETE cannot apply as e_1 : `any` would be a contradiction. The hypothesis is vacuously true.

If e_{before} is an UPDATE then the typing rule TUPDATE is used, as only UPDATE's with string literal indices can occur in the expression e_{before}. If the field s is not already present, then $e_{before} \longrightarrow e_{after}$ through EUPDATENOTFOUND. If the field is present, then, due to e_{before} being "strictly-typed", OBJECT and UPDATE operations are constrained such that if a field s is ever introduced, then $e_l[s] : t_{field}(s) = t_p$. Then, since the field type is consistent with the protection type, $e_{before} \longrightarrow e_{after}$ through EUPDATE. While EUPDATE introduces a cast, this must necessarily be an up-cast (allowed) as TUPDATE ensures that $e_r : t_r <: t_p$ (recall $t_p \neq$ `any`, due to the expression being "strictly-typed"). Since only up-casts are introduced, e_{after} is maintained as "strictly-typed", as desired.

If e_{before} is a CAST, then TCAST applies with $t_1 <: t_2$ (due to up-casts not being allowed in a "strictly-typed" expression). If t_2 is not a function type, then $e_{before} \longrightarrow e_{after}$ through ECAST and the expression remains "strictly-typed" trivially (as no problematic constructs are introduced). If t_2, however, is a function type, then $e_{before} \longrightarrow e_{after}$ through ECASTFUN, then a wrapper function is used and each argument of e is cast to the argument types of t_2, and the return value of e is cast to the return value of t_2. Since $e : t_1 <: t_2$, through SFUNC, these are guaranteed to be up-casts, and the expression remains "strictly-typed", as desired.

If e_{before} is a GET, then TGET applies as e_1 cannot have type `any`. Since e_{before} is well-typed, then e_1 has a component s of type $t_r = t_{before}$. Thus $e_{before} \longrightarrow e_{after}$ through EGET, which extracts this s component.

Theorem 5.2.4 (Progress)**.** *Given a "strictly-typed" (as defined above) expression $\Gamma \vdash e : t$, then if $e \longrightarrow^* e'$, then e' is a value. In other words, e does not get stuck.*

Proof: Progress follows from the Step-wise Progress lemma. Since the Step-wise Progress lemma ensures that a "strictly-typed" expression either takes a single step while staying "strictly-typed", or is a value, we can take this over an arbitrary number of steps to show that a terminating program eventually produces a value, and a "strictly-typed" e does not get stuck. ■

The Progress theorem implies that a well-typed expression that avoids the type `any` and applies a strategy for avoiding clashing protections (such as a global one type per field name policy, as in the definition of "strictly-typed", or only creating protected fields on objects for which the exact type is known) does not cause run-time errors. While, the "strictly-typed" condition has quite a strict restriction on updates, managing potential run-time errors that can occur in a well-typed program does not require as stringent constraints as presented, due to the inherently structured nature

of how updates are used in real LocusTypeScript programs. In LocusTypeScript, protected fields occur only in locus type declaration sites, and run-time errors can be avoided by having well-defined locus type hierarchies. For context, consider the `PlayerConnection` example: Assuming a fully typed expression, the only possible cause of unsoundness (run-time error) is if the `Connection` object already contains incompatible `player` and `sendName` members, and those members are not statically known. As long as no such assignments are possible, the resulting expression is guaranteed to be sound.

Theorem 5.2.5 (Locus Type Consistency)**.** *Given a well-typed expression $\Gamma \vdash e : t$, then if $t <: \{|L\}$, then necessarily $t <: \texttt{elaborate}(L)$.*

Proof: Locus Type Consistency follows from restrictions on both the AddBrand and Object expressions, the only expressions which can introduce a locus type L. Locus Type Consistency is enforced by only typing expressions where `elaborate`(L) are statically known to be present and L is to be added/included. ■

Due to Locus Type Consistency, a single check for an object brand implies all the type information in `elaborate`(L). As well, this invariant is necessary for locus types to have a predictable structure when used as types throughout the program that is richer than just a single type tag.

5.3 Imperative extensions

We extend the language presented with imperative constructs, illustrating how the functional core can be augmented to easily support the operations found in imperative languages. These constructs are analyzed at a high level during the resolution of `elaborate`, described in chapter 6. The additions presented here focus on introducing local mutable state. Global mutable state, like that in LocusTypeScript, can likewise be formalized in terms of this local state by passing state to every function and returning the resulting modifications. Object aliasing, another feature of LocusTypeScript, however, would require extensions to the core functional language. These extensions have been omitted as they are largely orthogonal issues, which would add baggage to the formalization, requiring some formalization of the JavaScript heap. The type preservation problems potentially faced by object aliasing are handled via run-time protection of object constraints. Furthermore, any program that is fully typed and has a global policy of avoiding protection clashes remains sound under aliasing, as the *Safety* rule remains applicable.

The SetVar operations allow for modifying variable bindings, making the local context act as if it were mutable. These bound variables are used exactly as normal

$$\frac{s \notin \{s_1..\}}{\texttt{set}\ s\ t\ e \longrightarrow \texttt{func}(s_1{:}t_1..)\{\texttt{return}\ \{(s_1{:}v_1|t_1)..\ (s{:}e|t)\ |\ \}:\{s_1{:}t_1..\ s{:}t|\ \}\}}\ [\textsc{DSetVarNotFound}]$$

$$\frac{}{\texttt{set}\ s\ t\ e \longrightarrow \texttt{func}(s_1{:}t_1..\ s{:}t)\{\texttt{return}\ \{(s_1{:}v_1|t_1)..\ (s{:}e|t)\ |\ \}:\{s_1{:}t_1..\ s{:}t|\ \}\}}\ [\textsc{DSetVarUpdate}]$$

$$\frac{\Gamma \vdash \mathit{stmt} : () \to \{s_1{:}t_1..|\}}{\texttt{eval}\ \mathit{stmt}\ e \longrightarrow \texttt{func}(s_1{:}t_1..)\{\texttt{return}\ e : t\}(\mathit{stmt}(\,))}\ [\textsc{DEval}]$$

$$\frac{\Gamma \vdash \mathit{stmt}_1 : t_1.. \to \{s_1'{:}t_1'..|\} \quad \Gamma \vdash \mathit{stmt}_2 : t_1'.. \to \{s_1''{:}t_1''..|\}}{\mathit{stmt}_1\ \mathit{stmt}_2 \longrightarrow \texttt{func}(s_1{:}t_1..)\{\texttt{return}\ \mathit{stmt}_2(\mathit{stmt}_1(s_1..)) : \{s_1''{:}t_1''..|\}\}}\ [\textsc{DPair}]$$

$$\frac{\mathsf{loop} = \texttt{func}(s_1{:}t_1..)\{\texttt{return if}\ (e)\ \{\mathsf{loop}(\mathit{stmt}(s_1..))\}\ \texttt{else}\ \{s_1..\} : \{s_1{:}t_1..|\}\}}{\texttt{while}\ (e)\ \{\mathit{stmt}\} \longrightarrow \mathsf{loop}}\ [\textsc{DWhile}]$$

Figure 5.3: The imperative desugaring

function/let bindings in the core language presented. To read the current variable result, we introduce the GetVar expression, and the Eval expression, which evaluates an expression in terms of a statement:

$$\begin{aligned} \mathit{stmt} \ &::= \mathit{stmt}_1\ \mathit{stmt}_2\ [\textsc{Pair}] \\ &\mid \texttt{set}\ s\ t\ e\ [\textsc{SetVar}] \\ &\mid \texttt{while}\ (e)\ \{\mathit{stmt}\}\ [\textsc{While}] \\ \\ e \ &::= \ldots \\ &\mid \texttt{eval}\ \mathit{stmt}\ e\ [\textsc{Eval}] \end{aligned}$$

The imperative language extensions are defined using a simple desugaring scheme to the base language. An imperative program consists of an Eval operation, which executes a statement and returns an evaluated expression. In this context, the "$\longrightarrow$" operator indicates 'desugars to', and is assumed to complete entirely before evaluation, using information from typing judgements. String literals $s_1..$ are used to signify members of an intermediate object holding the evaluation state, as well as the labels for intermediate function arguments. We implement the stmtoperations by desugaring them to a function that takes the previous state $s_1{:}t_1..$ and returns the changed result (wrapped in an object).

Minor syntactic leniences and points of ambiguity were used for brevity. We abuse syntax somewhat by assuming implicit unboxing of objects passed to a statement operation, such that passing an object $\{s_1{:}v_1|t_1.. \mid \}$ implies passing the arguments $v_1..$ in the correct order. Conversely, returning the values $v_1..$ is assumed to appropriately box the arguments into an object. The correct order of unboxing would be resolved during the desugaring phase. The unboxing and boxing operations are essentially boilerplate to allow for passing along local binding contexts as values, and their details are intuitive. There are ambiguities of how to apply types to the desugaring, reflecting different types that are possible in a sound system. For example, in WHILE desugaring, a uniform type must be assigned to the two possible results of a conditional. In LocusTypeScript, the union of the two possible results is used. We present how these and other imperative constructs are analyzed by LocusTypeScript in chapter 6.

Chapter 6

Locus Type Elaboration

We present the algorithm for computing the fields and brands in `elaborate`(L) of section 5.1, from the modifications made to a locus type variable. The locus type structure is determined from this variable, and implicit AddTag operations are added when the defining scope exits. The algorithm presented here applies broadly to imperative languages. The algorithm is presented using a simplified imperative language, with Update operations to a single object field. An extension to analyzing full objects is discussed.

6.1 Language abstraction

The algorithm performs conservative flow-based type refinement, assuming that all code paths are possible. As such, the features of the imperative language being analyzed are distilled to only a few relevant details needed for analysis of some particular object member. Loops and conditionals are simply represented as branches, as only the control flow details necessary for analysis are preserved. The only relevant detail for a block is whether it definitely occurs, or conditionally occurs. Similarly, all statements that change the type of the analyzed member are represented as an Update operation.

⟨*statement*⟩ ::= Update ⟨*type*⟩
| Branch ⟨*statement*⟩ ⟨*statement*⟩
| Block ⟨*statement-list*⟩
| Exit ⟨*number of blocks to exit*⟩

⟨*statement-list*⟩ ::= ε
| ⟨*statement*⟩ ⟨*statement-list*⟩

An UPDATE represents any operation that can change the type of the currently analyzed object member. This covers direct language constructs such as an assignment to the object member, as well as indirect constructs such as function calls that pass the object to a parameter with a locus type declaration.

A BRANCH abstracts any code-flow that causes mutually exclusive code paths to be taken. Any structured control flow maps trivially to one or more BRANCH nodes.

A BLOCK bundles a series of sequentially evaluated statements. BLOCKs exit naturally when the series of statements end, as well as from relevant EXIT statements.

An EXIT represents an imperative language control-flow jump statement such as JavaScript 'continue', 'break', or 'return'. It has an associated number indicating how many blocks down it exits from.

6.2 Overview

The algorithm presented here computes a single member of `elaborate`(L). The input to the algorithm consists of a function distilled to the abstraction presented here-in, with respect to some object member m. The distillation treats all blocks that occur 0 or more times as conditional BRANCHs, and all blocks that definitely occur at least once as unconditional BLOCKs. The output of the algorithm is a final type T computed for the member. Intuitively, for accepted inputs, the algorithm considers the permutations of each code branch (one branch or the other occurring) and exit (occurring or not occurring, note that unconditional exits are not modeled), and calculates T to be the union of the last assigned types in each permutation of control flow. Correct implementation of this is argued in Section 6.5, under *Union of all possible results*.

Certain inputs, however, are rejected (i.e., cause a static error) to avoid assignments that are considered error-prone. If two or more assignments exist in the same block and nesting level, each assignment must be either a subtype or supertype of the previous assignment. This prevents the first assigned type from being overridden by assignments that follow, preventing potentially inconsistent usage of the member. This limitation is applied only to assignments in the same nesting level and block to allow for conditional assignment to different types. Correct implementation of this is argued in Section 6.5, under *Disjoint type assignments*.

As an example, consider the analysis of an object member `obj.x` in a contrived function `gainsX`:

```
function gainsX(obj: declare HasX) {
    obj.x = 1;
    if (Math.random() > .5) {
```

```
        obj.x = "string";
    }
    while (Math.random() > .5) {
        obj.x = "string";
    }
}
```

The following intermediate form results:

```
function gainsX(obj: declare HasX) {
    Update number
    Branch // if
        Block {Update string} // obj.x = "string"
        Block {} // empty else case
    Branch // while
        Block {Update string} // obj.x = "string"
        Block {} // loop never occurs case
}
```

After calculating the union of the final assigned type after all possible permutations of control flow, the final type of `obj.x` is analyzed as `number` $\cup$ `string`. Note that the fact that the body of the while loop may occur multiple times does not change the result of analysis, and as such the `if` statement and `while` statement are represented identically.

As another example, consider the analysis of an object member `obj.x` in a function `gainsXIndirectly` that calls `gainsX`:

```
function gainsXIndirectly(obj: becomes HasX) {
    gainsX(obj);
}
```

Here, the following intermediate form results, after analysis of `gainsX`:

```
function gainsXIndirectly(obj: becomes HasX) {
    Update number∪string
}
```

6.3 Internal analysis state

The algorithm internally keeps a state tuple (T, E, A) used to compute the updated type after each program statement.

T represents the type previously assigned to the component (i.e, the field under analysis).

E is a list of pairs $(b,\ T_{exit})$ holding the type T_{exit} analyzed at an exit statement (e.g., continue, break, return in JavaScript). A block number b is associated, signifying how many nested blocks the exit escapes from.

A represents the assignability status of the analyzed object member. In the algorithm presented here, a simple rule enforces that two unrelated types cannot be assigned in the same block. This is an arbitrary heuristic, used to disallow direct member overwriting, which can occur easily from programmer mistakes. The A tuple member is used to track this. If A is true, the component can be locally overriden with any type. Otherwise, the overriding type must be a subtype of the current type, or vice versa.

Initially, the state tuple holds (`undefined`, ε, $true$), signifying that the object member is not yet assigned, the exit list is empty, and the object member is freely assignable.

6.4 Algorithm

The algorithm shown in figure 6.1 is defined as a function on the $\langle statement \rangle$ and $\langle statement\text{-}list \rangle$ grammar elements.

Rule (1) handles assignments and type changes due to function calls, T' is either a special error value or the assigned type.

Rule (2) handles branches, analyzing them both with the same state tuple, and returning a union of the results.

Rule (3) handles exit statements, adding a $(b,\ T_{exit})$ pair to the E list of pending exits, where b signifies number of blocks to exit from, and T_{exit} the type analyzed at that block.

Rule (4) handles block entry, starting analysis of the S statements with an empty E pending exit list, and A being true.

Rule (5) handles the block body, sequentially updating the $(T,\ E,\ A)$ tuple over the block statements.

Rule (6) handles the block exit. The list E has all its b values decremented, and elements with $b = 0$ are removed. T' is the union of the normal exit type with the removed T_{exit} values.

6.5 Proof of properties

Termination: Since all algorithm rules recursively destructure a finite tree the algorithm must always terminate.

$$\text{type } (\textsc{Update}\ T_a)\ T\ E\ A = (T',\ E,\ false) \quad (1)$$
$$where\ \ T' = \text{if } A\ \vee\ (T_a\ <:\ T)\ \vee\ (T\ <:\ T_a) \text{ then } T_a \text{ else error}$$

$$\text{type } (\textsc{Branch}\ s_l\ s_r)\ T\ E\ A = (T',\ E',\ A) \quad (2)$$
$$\begin{aligned} where\ \ T' &= \text{union } T_l\ T_r \\ E' &= \text{concat } E\ E_l\ E_r \\ (T_l,\ E_l,\ \_) &= \text{type } s_l\ T\ E\ A \\ (T_r,\ E_r,\ \_) &= \text{type } s_r\ T\ E\ A \end{aligned}$$

$$\text{type } (\textsc{Exit}\ b)\ T\ E\ A = (T,\ E',\ A) \quad (3)$$
$$where\ \ E' = \text{append } E\ (b,\ T)$$

$$\text{type } (\textsc{Block}\ S)\ T\ E\ A = (\ T',\ E',\ A) \quad (4)$$
$$\begin{aligned} where\ \ E' &= \text{concat } E\ E_{new} \\ (T',\ E_{new},\ \_) &= \text{type } S\ T\ \varepsilon\ true \end{aligned}$$

$$\text{type } (s_{head}\ S_{tail})\ T\ E\ A = (T'',\ E'',\ A'') \quad (5)$$
$$\begin{aligned} where\ \ (T'',\ E'',\ A'') &= \text{type } S_{tail}\ T'\ E'\ A' \\ (T',\ E',\ A') &= \text{type } s_{head}\ T\ E\ A \end{aligned}$$

$$\text{type } \varepsilon\ T\ E\ A = (T',\ E_+,\ A) \quad (6)$$
$$\begin{aligned} where\ \ T' &= \text{union } T\ (\text{map } E_0\ \lambda(b,\ T_{exit}).T_{exit}) \\ E_+ &= \text{filter } E_{dec}\ \lambda(b,\ T_{exit}).\neg(\text{equals } b\ 0) \\ E_0 &= \text{filter } E_{dec}\ \lambda(b,\ T_{exit}).(\text{equals } b\ 0) \\ E_{dec} &= \text{map } E\ \lambda(b,\ T_{exit}).(b-1,\ T_{exit}) \end{aligned}$$

Figure 6.1: Flow-based type refinement

Sequential handling: We show that the algorithm propagates assignments through the program tree sequentially. Observe that rules (2) and (4) simply recurse. As well, rule (5) computes T'' with the list tail from T' with the list head from T. This is the correct ordering. Other rules compute their type without recursion. Thus the program tree is handled sequentially.

Union of all possible results: We show that blocks consider the union of all possible exits as their final type. We consider each rule, showing they are correct in isolation:

(1) No effect on exits.

(2) Returns E', a concatenation of the previous E and the new E_l and E_r. Assuming that 'type' is correct over s_l and s_r, this rule is correct as all exits are considered.

(3) Appends a single exit. At this point, it has a correct b value by definition, and captures T_{exit} as the current type, as desired.

(4) Concatenates E_{new} to the exit list. It gives ε to the 'type' function, thus assuming the statement list is analyzed correctly, E_{new} should correctly be the exits defined in that block.

(5) Applies sequential application over statements. Since we conservatively consider all exits as conditional, the sequential consideration of every statement is the expected behaviour here.

(6) Always invoked exactly once per block, as a block exits. The assumed correctness of the other operations implies that we have a correct list of exits at this point, except their b values are off by 1. E_{dec} remedies this. Next, the $b = 0$ cases in E_{dec} are discarded and used to find the appropriate exit types. Inductively, since this scheme works for $b = 1$ exits in one block, it will correctly handle exits with arbitrary b.

Disjoint type assignments: We show that the A heuristic correctly only bars type over-writing within the same block:

(1) Correctly sets A to $false$, indicating only types with a subtyping relation can be assigned.

(2) Simply preserves A, as a BRANCH is considered a block for this purpose.

(3) Preserves A.

(4) Preserves A, and unconditionally starts the block with an A value of $true$, isolating it.

(5) Irrelevant to correctness: Enforces sequential evaluation.

(6) Irrelevant to correctness: the A value returned is not used in 4.

6.6 Extension to full objects

In implementation, we perform flow-based refinement over the entire defining object, as opposed to a single member at a time, as we require the entirety of the structure for typing judgements. Consider a variable that holds an object, annotated with a locus type declaration defining a type L. Let R be the set of intermediate references to that variable. To inform the type system from the host language, we must compute the final shape before the variable leaves scope (`elaborate`(L)), along with the intermediate shape of o for each reference $r \in R$. We compute whether the current object's type $t <:$ `elaborate`(L). [1] If that is the case, we insert an AddBrand operation, and the object is of type L.

While the `type` function is defined above for a single object member, it is trivially extended to analyze an object's member set and locus type brands. We track the current shape of the object with an object type from 5.1, $\{s_1{:}t_1 \,..\mid L_1..\}$. Update nodes can represent a call site from which the object o is known to have gained a locus type brand L_d. If the object o is passed to a parameter with a locus type declaration L_d, it gains the fields and brands of L_d (if valid). Unlike the single-field Update shown, however, this type information is discarded if the type is gained in only one side of a Branch.

The flow-based refinement algorithm is used in parallel with existing type inference to define types. The flow-based reasoning informs type inference, and vice versa. However, since flow-based reasoning can invoke more flow-based reasoning indirectly, some circular cases arise. These cases have been observed to be rare.

6.7 Handling recursive types

Consider if during analysis of a locus type T_A we arrive at a type T_B, which in turn uses components of type T_A. To avoid circular dependence, T_A instead must use a computed T'_B type, a version of T_B that does not depend on T_A. T'_B is itself computed with flow analysis, but treats members that exhibit circular dependence as untyped.

The flow-based refinement algorithm can be used in parallel with existing type inference to define types, with the flow-based reasoning informing type inference and

[1] Note, however, unlike the subtyping rule of section 5.1, field types may be subtypes of those in `elaborate`(L).

vice versa. However, since flow-based reasoning can invoke more flow-based reasoning indirectly, some circular cases arise. In the case that resolving the algorithm for a single member exhibits a circular dependency, the member should be considered untyped if its type is needed during its own resolution. While a more precise type may be possible to derive, leaving a value dynamically-typed is a pragmatic solution to this relatively rare case.

Consider the following (contrived) example of mutually recursive types `Chicken` and `Egg`:

```
function Chicken(this: declare Chicken; egg: Egg) {
    if (egg === undefined)
        this.cameFirst = "chicken";
    else
        this.cameFirst = egg.cameFirst;
}
function Egg(this: declare Egg; chicken: Chicken) {
    if (chicken === undefined)
        this.cameFirst = "egg";
    else
        this.cameFirst = chicken.cameFirst;
}
```

When, for example, `Chicken` is evaluated, the circular dependency is detected during analysis, and `egg.cameFirst` is computed using `Egg'`, an intermediate type computed assuming all `Chicken` members are untyped. Since `string` ∪ `any` is `any`, `Chicken.cameFirst` has type `any` (i.e., is untyped). The judgement `Egg.cameFirst : any` follows similarly. While an analysis of `string` would be ideal, deriving this type would require significant additional complexity. If a single annotation were to be added, for example, in `Egg`:

```
this.cameFirst =
  <string> chicken.cameFirst;
```

Then `Egg.cameFirst : string` would result regardless of analysis of `Chicken`, and `Chicken.cameFirst : string` would follow.

6.8 Variations and extensions

The algorithm model does not support statements such as C's '`goto`', which would require support for labeling code positions. However, the algorithm would treat it similarly to Exit, simply unioning with a stored type when a label is crossed.

The choice to disallow two assignments of unrelated types in the same block is arbitrary. This heuristic can be made more strict to disallow unrelated types in general. This strictness would suit type systems without union types, as there would always be a base type that covers both types, but not allow very common JavaScript use cases such as boolean $\cup$ string. Alternatively, to remain true to dynamic semantics, this heuristic can be simply dropped, at risk of not catching type errors.

Chapter 7

Performance

7.1 Performance

We measure the performance impact of locus types in LocusTypeScript. LocusTypeScript's only overheads arise from the installation of brands and type protection, neither of which are fundamental to locus types. With both of these features disabled, LocusTypeScript completely erases types, creating no overhead at all, so we measure the performance with these features enabled. For this experiment, we augmented a small number of programs with locus types and compared the result of running those on the V8 optimizing virtual machine and compared the program at 3 levels of run-time enforcement. The results are shown in Figures 7.1 and 7.2.

Firstly, the "Erased" level of run-time enforcement refers to locus types compiled with complete type erasure. The emitted code is just a type-erased form of the original code, making it equivalent to JavaScript.

Secondly, the "Reified" level of run-time enforcement refers to locus types compiled only with the operations required to support `declaredas`. The emitted code tracks the locus types declared for every object.

Thirdly, the "Protected" level of run-time enforcement refers to locus types compiled with full run-time protection, building on that of StrongScript. The emitted code installs getters and setters for protected fields.

As a baseline we used the benchmark suite provided by SafeTypeScript [16], which is in turn based on Octane 7. The benchmarks are crypto, navier-stokes, raytrace, richards and splay, presented in that order. These benchmarks were readily available in TypeScript, allowing for easy translation to LocusTypeScript. TypeScript ES6-style classes were translated to traditional JavaScript classes, annotated with locus types. Each benchmark times long-running iterative processes; iterations are performed before timing begins to allow the JIT a warmup period. We ran each

Erased		Reified		Protected	
runtime	**std dev**	**runtime**	**std dev**	**runtime**	**std dev**
383	0.5	382	0.5	635	2.2
392	0.9	389	2.4	788	7.6
37	0.4	61	0.3	3808	1.6
120	1	120	0.5	819	9.3
317	3.5	334	4.5	2186	21.5

Figure 7.1: Performance comparison of differing levels of enforcement.

Benchmark	**Reified slowdown**	**Protected slowdown**
`crypto`	1%	66 %
`navier-stokes`	0%	110%
`raytrace`	96%	10192 %
`richards`	0%	583 %
`splay`	5%	590 %

Figure 7.2: Performance comparison on the V8 VM. Times are in milliseconds, lower is better.

of these benchmarks 20 times for the three levels of enforcement. Performance for "Reified" is reasonable, given that it simply tracks a list on locus type objects. Performance problems for "Protected" largely stem from the installation of protection accessors. During tuning for the these bench marks, small differences were found to have profound effect on performance. The raytrace test was found to be particularly brittle, with dramatic slowdown with just an additional brand field being installed for every constructor. Extreme slowdowns occur with full protection due to the poor optimization of special fields in JavaScript. If locus types were supported at the level of the virtual machine, that cost would almost entirely vanish.

The benchmarks were run on Node.js 4.4.5, on an Intel i7-4770k with 16GB of RAM, running Fedora 20.

Chapter 8

Software

The implementation of LocusTypeScript was built on the StrongScript compiler as a practical means of focused work on locus types. While providing rich type concepts, the TypeScript compiler lacks facilities for types that change code semantics. These facilities are desirable for implementing sound locus types through run-time protection. StrongScript's system for run-time checks was amenable to our needs, and thus it was chosen as a base. The StrongScript compiler was updated to be in-sync with TypeScript 1.6, and thus the implementation fluently supports the majority of the features of TypeScript 1.6.

The implementation is open source, and can be found at **https://github.com/ludamad/LocusTypeScript**. The installation requires the use of "jake" build system, and can be built with the instructions listed at the provided URL.

To support programming in LocusTypeScript, the IDE API of the TypeScript compiler was extended. Using this extended API, a plugin for the `Atom` text editor was created, based on the `atom-typescript` plugin. The plugin can be found at **https://github.com/ludamad/atom-locustypescript**, with installation instructions provided. With this plugin, a fully featured IDE for programming in multifile LocusTypeScript programs is made available, with refactoring options for turning locus types into conventionally explicit types.

Chapter 9

Conclusions and Future Work

Locus types are a new technique for declaring types in a syntactically-minimal way. By marrying existing flow-based type refinement with existing gradual typing, much of the tedium of explicit specification can be eliminated. We believe that locus types fit dynamically-typed programming languages well, as "code is types". Locus types embody intuitive notions of type, as they type objects based on code sites they have flowed through. Locus types can be used to type objects created with a constructor, akin to traditional classes, as well as to express types with a temporal meaning such as "objects that have passed some security check". Since enforcement depends on techniques already available in gradually-typed languages, locus types are usually semantically equivalent to explicitly-specified types, and so can be used transparently in existing systems. We believe that locus types provide a useful stepping stone in gradually-typed languages, making the declaration of types easier on dynamic-language programmers, and are relatively uninvasive for implementers.

At present, LocusTypeScript's error messages are essentially StrongScript's error messages, with some additional errors about incompatible types. Because locus types are derived from type refinement, these error messages could be made much more expressive. Every type error involving a locus type can refer to the exact line(s) of code which caused the refinement engine to infer the incompatible type. Because these errors would allow programmers to compare the types inferred to the relevant parts of code, we believe they could be more useful than the simple type errors already provided.

The design of locus types as presented highlighted concerns found in JavaScript. However, it is likely other languages have their own unique considerations for locus types that can be explored. Likewise, alternate designs for defining types using flow-based type refinement in JavaScript could also be explored.

References

[1] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP 2014–Object-Oriented Programming*, pages 257–281. Springer, 2014.

[2] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn—robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009. doi: 10.1145/1639949.1640098. doi:doi: 10.1145/1639949.1640098.

[3] G. Bracha. The Strongtalk type system for Smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*, 1996.

[4] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993. doi: 10.1145/165854.165893. doi:doi: 10.1145/165854.165893.

[5] Facebook. Flow language documentation. **http://flowtype.org/docs/**.

[6] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA*, 2001. doi:doi: 10.1145/504282.504283.

[7] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002. doi:doi: 10.1145/581478.581484.

[8] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676992. URL **http://doi.acm.org/10.1145/2676726.2676992**.

[9] A. Guha. *Semantics and Types for Safe Web Programming*. PhD thesis, Brown University, 2012.

[10] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Programming Languages and Systems*, pages 256–275. Springer, 2011.

[11] T. Jones, M. Homer, and J. Noble. Brand objects for nominal typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[12] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508175. URL **http://doi.acm.org/10.1145/2508168.2508175**.

[13] Microsoft. Typescript – language specification version 1.8. Technical report, Jan. 2016.

[14] G. Nelson. *Systems programming with Modula-3*. Prentice-Hall, Inc., 1991.

[15] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103714. URL **http://doi.acm.org/10.1145/2103656.2103714**.

[16] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe and efficient gradual typing for TypeScript. In *POPL*, 2015. doi:doi: 10.1145/2676726.2676971.

[17] G. Richards, F. Zappa Nardelli, and J. Vitek. Concrete types for typescript. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[18] J. Siek. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[19] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007. doi: 10.1007/978-3-540-73589-2_2. doi:doi: 10.1007/978-3-540-73589-2_2.

[20] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837630. URL **http://doi.acm.org/10.1145/2837614.2837630**.

[21] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, 2006. doi:doi: 10.1145/1176617.1176755.

[22] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. In *POPL*, 2008. doi:doi: 10.1145/1328438.1328486.

[23] D. Unger and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1987.

[24] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 310–325. ACM, 2016.

[25] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *DLS*, 2014. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661101. URL **http://doi.acm.org/10.1145/2661088.2661101**. doi:doi: 10.1145/2661088.2661101.

[26] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[27] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *ACM Sigplan Notices*, volume 45. ACM, 2010.