Pluggable, Iterative Type Checking for Dynamic Programming Languages

Tristan Allwood (toa02@doc.ic.ac.uk)

Professor Susan Eisenbach (supervisor)
Dr. Sophia Drossopoulou (second marker)

University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Project sources available at: `/homes/toa02/project/fleece/`

June 13, 2006

Abstract

Dynamically typed programming languages are great. They can be highly expressive, incredibly flexible, and very powerful. They free a programmer of the chains of needing to explicitly say what classes are allowed in any particular point in the program. They delay dealing with errors until the last possible moment at runtime. They trust the programmer to get the code right.

Unfortunately, *programmers are not to be trusted to get the code right!* So to combat this, programmers started writing tests to exercise their code. The particularly conscientious ones started writing the tests before the code existed. The problem is, tests are still *code*, and tests would still only point out errors at runtime.

Of course, some of these errors would be silly mistakes, e.g. misspelled local variables; some would be slightly more complicated, but a person reading the code could notice them. They are errors that could be detected while a program was being written. Static type systems could help detect these mistakes while the programs were being written.

Retrofitting a mandatory static type system onto a dynamically typed programming language would not be a popular action, so instead the idea of *optional* and then *pluggable* type systems was raised. Here the type systems can be applied to the language, but they don't have to be. Mistakes can be caught statically, but the good features of the language, expressiveness and flexibility are not compromised. Unfortunately, they are just an idea.

This project explores the notion of pluggable type systems for dynamically typed programming languages. It also looks at how these type systems could interact with each other as parts of the pluggable type system improve or are updated. It isn't all just an idea anymore, a real application has been developed that demonstrates how these pluggable type systems could work on the code of a simple dynamically typed language.

Acknowledgements

I wish to acknowledge and thank my supervisor, Professor Susan Eisenbach for all her help and guidance on this project; and for providing pointers to the papers that have subtly moved the direction of my project to where its final form.

Thanks must also go to my second marker, Dr. Sophia Drossopoulou, who has pointed out areas to which I needed to give further thought and given improvements to the presentation of $\mathcal{R}_{sub}$.

I also wish to thank my peers in the Department of Computing at Imperial, particularly Marc Hull, Daniel Burke and Matthew Sackman. Conversation, discussion and debate with you all has led to me forming a more rounded view of this area of theory, highlighted interesting issues I would otherwise have ignored, and an appreciation for other points of view.

A big thank-you must also go to the members of the SLURP reading group in the department who gave feedback on a presentation of this project

Many thanks to Doreen Wright for printing and Jon Wright for providing commentary upon Deal or No Deal.

Finally I wish to thank my family for all their love and support, without which this project would not be possible.

Contents

Chapter 1

Introduction

This is the Report for my Final Year Project, *Pluggable, Iterative Type Checking for Dynamic Programming Languages*. To explain the sheep on the cover page, the application written to accompany the project is entitled *Fleece*.

1.1 Context

The theory of type systems for programming languages is well established, as are the benefits they bring. Type systems can give an abstraction of program code being executed, and then guarantee properties of the code. This analysis can be done before any code is executed, rejecting programs where the required properties cannot be proven; or it can be done at runtime, raising a predictable error condition should the type-properties be broken. This rejection means that the possible programs generated from a given syntax is restricted to those that exhibit (or are free from) certain behaviours.

Type systems are often seen as checking that variables belong to certain sets of values (in the JAVA world sets of classes or interfaces). However, many different type systems are the study of active research and development, and could offer many different types of analysis and guarantees to programming languages.

The type systems of programming languages also have other uses, for example providing machine-checkable documentation to the programmer, and for guiding possible compiler / interpreter optimizations. Many of these advantages are traditionally brought by *mandatory*, *static* type systems, where the checking is done in a pre-runtime phase. Languages designed with *dynamic* type systems tend not to have these properties available[1]. However, mandatory, static type systems are often seen as brittle and restrictive to the expressiveness of the language concerned.

Recently, Bracha ([Bra04]) has suggested *pluggable* type systems, which, he argues, should be able to

> provide most of the advantages of mandatory type systems without most of the drawbacks.

[1] Of course this does not mean a static type system or checking cannot be retrofitted onto them. For example STRONGTALK [Str] or STARKILLER [Sal04]

It is this idea of pluggable type systems that this project is concerned. If multiple type systems can be used on a language, is it possible that they could interact, and provide more information than they could individually? If so, how could this be done?

1.2 Project Contributions

Within the context given above, this project makes the following contributions:

- A restricted subset of the object-oriented dynamically typed programming language RUBY has been formalised. It is named $\mathcal{R}_{sub}$ (chapter 3).
- A motivating example describing pluggable type systems for $\mathcal{R}_{sub}$ has been given (chapter 4).
- A way of viewing pluggable type systems such that the type systems can interact in a compositional manner has been presented. With this description, a type checking algorithm is given that will terminate (chapter 5).
- A discussion is given on the possible ways of extending the type checking algorithm for incremental program development. This is useful for reusing type information that does not need refreshing across edits to a program as it is being developed.
- An application has been developed that allows programs to be constructed in a controlled incremental fashion. (chapter 6).
- The type checking algorithm has been implemented in this application, with some of the incremental type checking extensions discussed. The information the type checking algorithm annotates the program with is visible within the application.
- $\mathcal{R}_{sub}$, and some of the example type systems described have been implemented within the application.

1.3 Motivation

This project was motivated by wanting to take a step towards providing solutions to the following problems:

Software may need to be deployed in many different environments and may need to be provable to conform to different requirements. For example, a company may only want to deploy software that it can show has a certain property (perhaps doesn't use disk access). Having pluggable type systems in a language means that the company can create its own type system to assert this property and then apply it to the source code to check it obeys it.

The APIs that developers have to work against are often large and complicated with imprecise documentation or convention dictating their use. There is little help within existing tools for statically telling a developer they are incorrectly using a method, for example. With precise contracts representable by the type system, and type analysis tools working to a well-defined API, information can be moved and presented in a uniform manner to the developer by their programming environment.

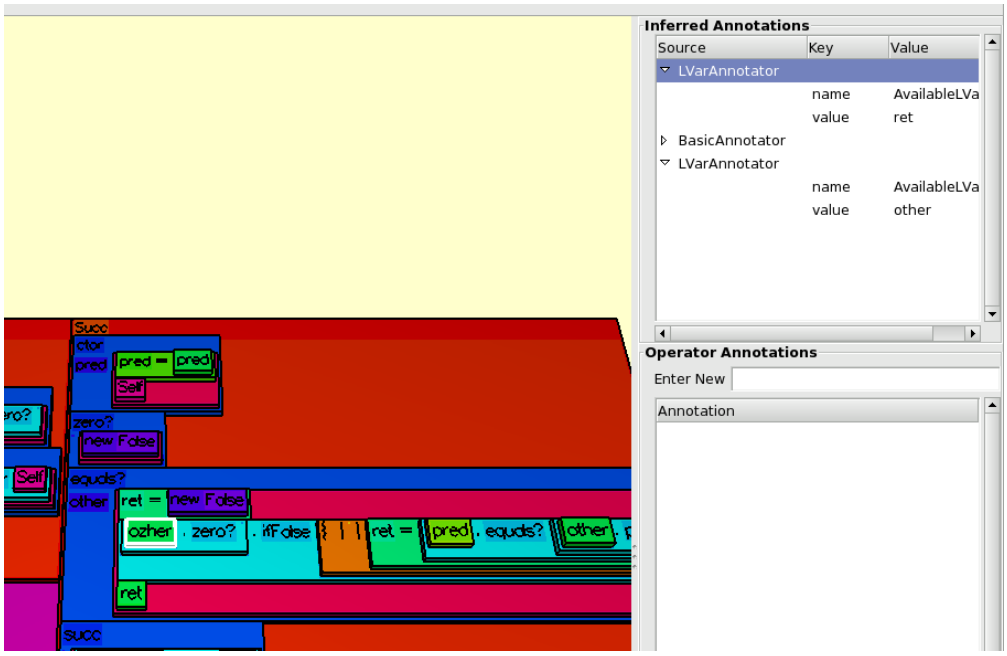


Figure 1.1: A screenshot of the FLEECE application that was developed as part of this project. The boxes are the abstract syntax tree of a program. The box in white is in error, it is trying to reference a local variable that is not available to it (as indicated by the inferred `AvailableLVar` annotations)

Developers spend much of their time thinking when working, during which time the computer is (generally) idle. These spare CPU cycles could be going to good use. Type inference would ensure that the CPU is active during the developer's thinking time and also would save the developer from telling the computer redundant information, saving them even more time to think.

Programming languages develop, change and evolve and are usually specified by their implementation as opposed to formal models or precise specification. As such any mandatory type systems which may be used with the language may evolve making programs that once typed not type any more. Also problems in the type system require the whole language to be redeployed, coupling the two together. It is more likely, however, that the language would not be redeployed. Decoupling the type system from the program itself allow the two to grow independently.

1.4 Report Organisation

This report is organised as follows. Firstly the relevant background material is reviewed in chapter 2. The working example language that this project has used is introduced and formalised in chapter 3, before examples of how pluggable type systems might work on it in chapter 4. The theory of these pluggable type systems is then formalised with an algorithm for how type checking would work in such a system in chapter 5. These ideas were implemented in the application FLEECE, which is the subject of chapter 6. Finally the success of the project is discussed in chapter 7 before some final conclusions are drawn and suggestions are given to possible future work in chapter 8.

Chapter 2

Background

2.1 Introduction

This part of the report gives an overview and discussion of the relevant literature to this project. section 2.2 gives a review of the papers that inspired and influenced the project. Dynamically typed programming languages are discussed in section 2.5, and relevant type system theory in section 2.6.

2.2 Project Roots

The following papers where the main sources of inspiration and relevant background for this project. Following the review of the papers, other projects are outlined that are related to the context of this project.

2.2.1 Pluggable Type Systems

The name *pluggable type system* seems to have originated with Bracha in [Bra04]. In this paper Bracha outlines the case for, and what he means by them. This is an extension to the idea of optional type systems, whereby the language in question is not influenced by the type system for its runtime semantics. Allowing multiple type checkers to be "plugged in" to the language at compile time means that most of the advantages provided by mandatory, static type systems can be regained without introducing the problems such mandatory systems have.

Bracha identifies four of the main, commonly understood, advantages to mandatory, static typing:

- Providing machine-checkable documentation
- Providing conceptual frameworks that aid the programmer
- Providing early error detection
- Providing opportunities for optimization based on type information

These can all (at some level) be preserved by optional and pluggable type systems; for this project I aim to focus mainly on the third property, with the first also being relevant.

However, Bracha argues, in mandatory systems you suffer a constraint on the expressiveness of the language and, more importantly, can become dependant on a system which may later prove to fail. Mandatory type systems can easily become depended upon for security (or other) guarantees and they have been shown to fail in the past.

Bracha also points out that the formalization of type systems is usually upon an abstracted or idealized model; should the model miss an important detail the guarantees it offers are lost. Furthermore, it is not impossible for implementations to be buggy, and with new additions to languages or their type system interacting in exponential numbers of different ways, problems are bound to arise[1].

An *optional type system* is defined by Bracha as one that:

1. has no effect on the run-time semantics of the programming language
2. does not mandate type annotations in the syntax

Emphasis is placed upon the first requirement. It also has the consequence that some programming paradigms (public fields, class based encapsulation and static type based overloading) become either prohibitively expensive or just impossible. However, as Bracha says,

> The above features are all things one is better off without anyway.

With optional typing, it is possible for a type system to evolve at a different (faster) pace than with a mandatory system. Programs that would run but wouldn't type check can have an improved type-checker validate them and they will still be able to run as before.

Bracha also points out that the evaluation rules of the lambda calculus are (usually) unaffected by any type system applied to it. The type systems simply reject certain classes of programs.

With the argument in place for optional type systems, Bracha then considers what would happen if we moved the type-checking out of the language specification and made the language capable of supporting various type-checkers specialized for different purposes. To do this, the AST of the language would need to support (almost) arbitrary metadata for the various annotations needed by the type systems. This then moves into the area of research; languages such as JAVA and $C\#$ support annotations, but not at the statement or expression level.

There is also a small discussion on *type inference*. Bracha makes the point that type inference should be optional, like the type checking. Possibly inferencers can be used to decorate the program AST with annotations that the programmer would otherwise have to declare himself. Also inferencers could be tools in an IDE (like auto-complete) to speed up the annotation process for the user.

2.2.2 Static Typing Where Possible, Dynamic Typing When Needed...

This ([MD]) is an overview paper that puts forward the argument for seeking the "golden middle way between dynamically and statically typed languages." This seeking is done by exploring

[1] The Java Bug Parade is a good demonstration of compiler (and other) problems in a mature language that has recently undergone a lot of expansion [Sunb].

what the authors believe developers really want when they say they need static or dynamic typing. The overruling theme from these ideas is (as in the title of the paper) that static typing should be used where possible, but dynamic typing is necessary when needed.

The introduction of this paper overviews the arguments for and against using just static and/or dynamic typing. What is most interesting for this project is the observations of the major disadvantages of both;

> Static type checking is a compile-time abstraction of the runtime behavior of your program, and hence it is necessarily only partially sound and incomplete.
>
> Defending the fact that all type-checking is delayed until runtime is a good thing, is playing ostrich tactics with the fact that errors should be caught as early in the development process as possible.

When Programmers Say "I Need Dynamic/Static Typing", They Really Mean...

According to the authors, when developers say they need dynamic/static typing they really mean they want any of certain language features that commonly used static/dynamic languages provide. This list includes type inference, contracts, coercive subtyping, Generics, unsafe covariance, ad-hoc relationships with prototype inheritance, lazy evaluation, and first class support for the various (ab)uses of the `eval` keyword/function. For this project, the most interesting of these are the first two.

I want type inference

Developers from a statically typed background may be used to writing out full type declarations for all variables (especially local variables). While this is not a requirement of statically typed languages[2], modern mainstream languages[3] do make the less verbose dynamically typed languages look appealing. However this is asking for the wrong thing, as the authors point out

> Not requiring programmers to write types as dynamic languages do is great; but not inferring the types of these variables whenever possible is literally throwing away the baby with the bath water.

The authors also make an interesting observation, in that static type information (inferred or otherwise) plus code can always be used to seed a fully explicit dynamically typed version of the code. With type inference and static type information it is then possible to be more concise than having just a dynamically typed language.

I want contracts

As we have already noted, static typing can prove only the absence of certain errors statically, but not all. What some developers really want is the ability to specify more precise contracts and have them checked as much statically as possible, with predictable checking at runtime for when this is not possible. In a manner very similar to Flanagan's ([Fla06]) the authors describe:

[2] It is actually a requirement of manifest typing - see Section 2.6.2 for definitions
[3] Read JAVA, C++

> The compiler should verify as much of a contract P for some expression e as it statically can, say Q, signaling an error only when it can statically prove that the invariants are violated at some point, and optionally deferring the rest of the checking $Q \implies P$ to runtime via the witness f:
>
> $$\frac{Q(e) \rightsquigarrow e' \qquad Q \implies P \rightsquigarrow f}{P(e) \rightsquigarrow f(e')}$$

The authors also show that coercive subtyping (downcasts and extensions thereof in JAVA) can be modelled as contracts; showing that this is a very flexible hybrid type checking technique.

Summary

This is a provocative paper that suggests many ideas for what future language developers should take with them in terms of features and abilities. It also argues the case for integrating static and dynamic type checking in a hybrid way. However the argument differs from Bracha's ([Bra04]) in that there is no focus on separating the type systems from the language, instead there is the implicit assumption that a language should be developed with a static/dynamic integration deliberately put in.

2.2.3 On the Revival of Dynamic Languages

This paper [NBD+05] argues that current programming languages are inherently too static to be good tools for the constantly evolving environments that their programs have to reside in. What the authors believe is necessary is the development of new programming languages that are designed around supporting change at run-time. After a short general introduction, they then focus upon 5 different areas of potential research that together could be used to develop their new class of "dynamic" languages.

The scope of this paper is wider than my project, indeed my project falls neatly into a single one of their potential research tracks - *Pluggable Types*. Although this section is fairly short and not technically detailed, they do make some interesting observations.

Firstly they re-iterate the point that static type systems will always reject some correct programs because they cannot prove them correct. However they go on to call static type systems "the enemy of change". This is mainly due to the inherently dynamic nature of reflexive code, which causes workarounds to be cumbersome and verbose.

The central belief that the authors hold with regard to type systems is one shared by Bracha [Bra04];

> A type system should never be used to affect the operational semantics of a programming language.

They then cite Bracha's STRONGTALK[4] [BG93] as an example of an *optional type system* that satisfies this property.

[4] Described in Section 2.2.4

They also make the point that there are many different types of type analysis under ongoing research, alias types, confined types, scoped types etc. and that it is unrealistic to expect new statically typed languages to take advantages of these idioms. However it is more reasonable to expect a new dynamic programming language to be developed that can take these type systems as plug-in extensions.

They also cite another example, of a type system that can be used to give hard real-time guarantees about performance and execution times.

2.2.4 Strongtalk: Typechecking Smalltalk in a Production Environment

This paper ([BG93]) describes the Strongtalk typechecker that was designed and implemented for a SMALLTALK dialect. It is a fully optional type checker, designed to be downward compatible with SMALLTALK, and it uses many important concepts (such as separating an object's type from class). The type checking also does not rely on non-local code analysis, which means it can be used incrementally.

Type System

This project demonstrates pluggable type systems for an Object-Oriented core language that is similar to RUBY (which is in turn similar to SMALLTALK). STRONGTALK is one example of such a type system and thus the ideas and design decisions are relevant.

The type system has the following features:

- It is purely structural (as opposed to nominal); subtypes and subclasses have separate lattices. Type identity is preserved using *brands*.
- It has parameterized types and classes.
- It has polymorphic messages, with a mechanism to allow type inference of the type parameter.
- It distinguishes inheritance from sub typing.

Design Notes

The purely structural type system of STRONGTALK means that an object can be used anywhere if it statically supports the correct *protocol* of messages and signatures. These protocols can be user defined, but also every class definition implicitly defined a protocol that its instances support.

STRONGTALK does not rely upon type inference. The argument for this decision is based upon source-level annotations being useful for human-readable documentation; and that an enhanced source browser could use inference to place the annotations on the tree. However (apparently) inferred structural types can quickly become large, complicated and unwieldy for human use.

The paper also spends some time discussing the merits of local vs. non-local code analysis. Although non-local code analysis is in some regard more powerful / complete, it does however make the type analysis less modular and non-incremental. An example given is that the changing of method body in a superclass will require re-checking the uses of that method in the context of

all subclass bodies. This requires the source for all subclasses to be available and known which is a heavy requirement.

In the STRONGTALK type system, a subtype is always fully substitutable for a supertype. This means that being a subclass does not necessarily make something a subtype. As is noted by the authors, it is sometimes convenient for a programmer to define an incompatible subtype for use in a controlled way.

An interesting note is that the definitions of instances and their classes are always mutually recursive in SMALLTALK, and therefore the protocol definitions in STRONGTALK are too. This is due to all instances having the message `class` to return their metaclass, and all metaclasses having the message `new` to create a new instance. Inheritance may cause either of these protocols to change, and will hence induce a change in the other.

The authors remark that a possible risk with structural typing is the possibility for syntactically equivalent objects to be used semantically incompatible situations. To combat this, the STRONGTALK type system has the notion of *brands*; Brands are tags given to protocol definitions, and when a class is declared as supporting a given protocol, it inherits that brand. STRONGTALK also verifies that a class declared as implementing a branded protocol does so (type) correctly.

STRONGTALK's type system allows union types. This means that, for example, the `head` method of a generic `List[T]` would be typed to return `<T | Nil>`. The type system would then require the programmer to check at runtime if they got back what they expected.

When defining parameterized methods, the developer can quite precisely specify where the parameterized type comes from. Keeping with the generic `List[T]` example, the `map` function could be defined as:

```
map: <Block[T, ^S]> ^<List[S]>
    where S :: (actual arg:1) returnType
```

The `where` declaration means that STRONGTALK can correctly infer type parameters at call sites without requiring a sophisticated type inference algorithm.

Experience

The authors used STRONGTALK to write a significant amount of code (including a STRONGTALK implementation) and based upon their experience of this made a few comments.

The first thing the authors note is that the fact that SMALLTALK / STRONGTALK has (and supports) first-class code blocks, and that these interact well with the type system providing a lot of useful idioms that STRONGTALK can easily type. For example, control structures, call backs and simple exception handling can all be done simply with blocks.

They also note that in practice it is useful to explicitly define a separate protocol if it is likely that more than one implementation of it will exist. The alternative of using the implicitly defined protocol of a class can become confusing when a second implementation refers to it. This is analogous to programming JAVA using interfaces for all type declarations and classes just for implementation.

Additionally, because STRONGTALK does not require access to the implementations of referenced classes, it is possible for the standard library to have type declarations given to it, even in cases where the source would not type check. This is a very strong advantage of designing type systems that require only local source code analysis.

2.3 Other Relevant Work

Many programming languages have integrated development environments (IDEs) available for them, to aid the developer experience while programming. These can vary in complexity from being alternate modes to a text editor, to fully-fledged highly re-configurable and plug-in extensible applications.

IntelliJ IDEA ([Jet]) is a mature, commercial IDE for JAVA. It boasts over 500 different ways of statically analysing source code that the developer is working on for problems, varying from coding style to portability bugs. The latest version can also work with JAVA 5 annotations ([Suna]), allowing the developer ways of creating new assertions that IDEA checks during the coding phase.

For example, the `@Nullable` or `@NotNull` annotations can mark a JAVA method as possibly returning (or definitely not returning) `null`. IDEA can pick up on these annotations and check them for the user, and also track variables assigned from annotated methods, warning if a method call is made without a null check first if necessary. Interestingly, JAVA 5 annotations used in this way can be used to replace (or plug-in) functionality that was added explicitly to other JAVA variations. For example the NICE programming language ([Bon]) had explicit constructs to mark variables as possibly `null` or not.

It could be seen that JAVA annotations, with appropriate analysis tools could provide some level of plug-in enabled type-checking. While this is true at the class and method signature level, it is an unfortunate limitation of annotations that you cannot annotate at the statement level.

There have been some *Design By Contract* (DBC) tools and languages (most notably, EIFFEL [Mey92]) that fall into a related field as this project. For example the JAVA MODELLING LANGUAGE, JML ([LC05]) allows the assertion of pre and post-conditions on code, as-well as invariants. These can be fairly involved, and involve quantified assertions. It also supports behavioural subtyping, allowing pre, post, and invariant conditions to be specialized (or generalized as appropriate) in inherited specifications. JML can check contracts at both run-time and statically , and has a plethora of extension tools developed for theorem proving, invariant discovery and others.

In order to increase the expressivity of the contracts developed, JML requires that methods to be used in contracts are side-effect free and labelled by the developer as *pure*.

Dependant type systems allow a middle-ground between contracts and type systems; with them (depending on the expressiveness of the dependence) the type system can express the structure of the output in terms of the structure of the input(s). DEPENDANT ML ([XP99]) allows one such variant of dependant types, where the dependence may be a natural number, and thus can reason about (for example) the size of lists going into and returning from functions. This can be done statically. Flanagan's Hybrid type system ([Fla06]) uses dependant types, but an non-decidable variant allowing a lot more expression at the cost of the ability to statically check the code.

It is worth noting that the distinction between precise typing and classical control-flow analysis can become blurred. There has even been work ([Hei95]) to draw an equivalence explicitly between the two.

There has been a body of research looking into the inference of types in SMALLTALK. Suzuki ([Suz81]) approached the problem from a data-flow point of view, and used a non-local analysis. A variable's type was taken to be the union of all the possible classes it may refer to. Borning and Ingalls ([BI82]) developed a statically typed variant of SMALLTALK, allowing an inference procedure to determine the types of local variables. Here, they take "type" to mean (in the default)

the class of the variable in question, but also allow parameterizable types and possible extensions into other type schemes.

A more novel piece of work has been done by using the test-code of an application, along with an instrumented runtime system to dynamically infer the types of instance variables and method arguments ([RBFDD98]). This work can also find many other interesting properties of the program (for example dead code), but it has a reliance on the test-suite used being complete.

Aycock ([Ayc]) has explored using type inference to convert PYTHON code into PERL code. As PERL variables are typed (in the sense they must be one of `$@%&*`) the type inference is necessary as PYTHON variables are untyped. He notes, however, that even when making some very simplifying assumptions (ignoring dynamically evaluated code, and performing flow insensitive analysis of local variables), it is possible to get reasonable results. There is also a strong argument, backed up with some statistical data, made for the claim that:

> Giving people a dynamically-typed language does not mean they write dynamically-typed programs

The Object-Oriented language this project uses as an example dynamically-typed language (RUBY) does have dynamic capabilities, but for simplicity they have left them out of the subset that is chosen to use.

Part of this project has been to develop a graphical tool to allow the construction and type checking of a program. Whilst the notion of an IDE is not new for working with and editing code, there are also languages that only exist in a visual world. While many of these are workflow languages, or meta-representations for other languages; there are some that are languages in there own right. Ongoing work by Edwards and his language, SUBTEXT, ([Edw05]) demonstrates that it is possible to have an entire language in the GUI, much as an entire SMALLTALK or SELF application and editor lived in an executable image.

2.4 Discussion

Bracha's idea of decoupling the type system(s) from the language definition / specification was the starting point for this project developing a core language and then evolve analysises upon it. While this decoupling is not novel (many experimental type analysises have been built for existing programming languages), no generic framework for working with the AST of the language in a manner that isn't tied to a specific language has been found.

There are many object-oriented "scripting" languages that are enjoying popularity at the moment (for example RUBY [Matb], PYTHON [vR], JAVASCRIPT [Net], PHP [GS] to name a few). Many of these languages are dynamically, structurally and strongly typed. This means they have defined semantics at runtime, and do not require (and in many cases do not have) static type analysises existing for them. They would provide good candidate platforms for pluggable, hybrid type systems to reside upon / be developed for. This project has chosen one these languages (RUBY), and formalised a subset of it for developing type systems against.

Type inference algorithms can be used to automatically generate type requirements and to propagate the constraints as necessary. This can be done without developer intervention (but may be refined if the developer requires it), and can save any tedious work. For example invoking a

method on a given argument implies the argument should have that method defined. Type inference algorithms and strategies are things that the framework developed in this project is able to support.

The more general survey papers reinforce the general principals underlying this project. The astute observation that some developers want to use dynamic programming languages simply because they do not need to type their variables makes a good case for type inference in general. Also the mantra

> A type system should never be used to affect the operational semantics of a programming language

I believe to be important to hold onto. Since any entirely-static type system will always be a limitation on the expressivity of a programming language it makes sense to decouple them.

STRONGTALK, which was a successful implementation of an optional type system on top of SMALLTALK (which is a language very close to the target one for this project) demonstrates that it is possible to create an incremental type system that keeps much of the expressivity of the original language. The fact that the subtype and subclass lattices can be split and seemingly not complicate matters too much will also be something to explore in my project.

The JAVA programming language is well established as an industry strength Object-Oriented language. The central argument of a few of the earlier papers, that static type systems can be brittle and harm expressivity is demonstrated by the fact that JAVA 5 was released with a more powerful type system. That wildcards were developed almost especially for the purpose drives even more how important it is for type systems for programming languages to evolve. However, since the (original) type system was mandatory in the VM, and the language must be kept backward compatible with this type system, the casts that generics should have removed have been kept in (albeit hidden from the developer).

Finally, there is a lot of related work in many areas relating to this project, from general type inference for SMALLTALK, to adding contracts to JAVA. This project then can be seen as bringing together of several different fields of research that often stay disjoint. The boundaries between static and dynamic checking, control flow analysis and type systems, developer tools and programming languages become blurred in a novel and exciting way.

2.5 Dynamically Typed Programming Language Features in RUBY

Common dynamically typed[5] languages are not designed with a static type system in mind, and usually have rich semantics for handling error cases at runtime. In these cases, adding a sound (but incomplete) type system to the language would reduce the number of otherwise valid programs that could be expressed in the language. A complete but unsound type system would aid a developer by disallowing programs that will certainly fail at run-time, but not hurt the expressibility of the language.

A feature of the dynamically-typed programming languages under consideration is that any syntactically valid program in the language will have an associated semantic meaning when executed (even if it throws an error). That is to say any program instance that can be generated from the language's BNF (Backus-Naur Form) / grammar can be run.

[5] As pointed out in [Pie02], the word *typed* is a misnomer, it should really be *checked*.

This project uses a subset of the RUBY programming language as an example for demonstrating how pluggable type systems could work. RUBY as a language has a very consistent view that "everything is an Object". It also has no static type system to speak of, and has good support for meta-programming. The targeting (a subset of) an established language mean that the interpreter already exists for the language. On an interesting note, for the next version of RUBY, there has been raised the possibility of adding optional type systems into its VM ([Mata]).

This section will discuss particular aspects of the RUBY language that are interesting from a typing point of view. It does not aim to be a general introduction to the RUBY programming language; if more detail is required I refer the reader to [TH01] as an excellent reference.

2.5.1 Classes and Modules

In Object-Oriented languages, there is usually a well-defined mechanism for producing new objects. There are several ways of doing this, for example by copying an existing object[6]. However in RUBY new instances are created from objects that prescribe their structure. These object prescribing objects are called *Classes*.

Classes have a method `new` that creates an instance of that class. Any methods that are defined on the class are accessible from that instance. Classes can also `extend` from other classes (by having a superclass), forming a class hierarchy. This hierarchy is used when resolving method calls on the instance. A superclass is searched for a method when it can't be found in the current instance's class. In RUBY a class can only have a single superclass[7]

Of course, there may be times when you want to inherit implementation from more than one place. In this case, RUBY also allows *module*s to be `include`ed into a class. A module is almost identical to a class, except it does not have a superclass or a `new` method. A common use of a module is to give derived capabilities to a class, for example all the common comparison operators / methods (<, <=, >, >=) can be derived from the definition of the compare method/operator (<=>). To save writing out these derived definitions in every class that supports comparison, the `Comparable` module can be included in the class.

From a typing point of view, classes and modules can be quite interesting. Modules can be seen to have an interface of expected methods (and instance variables) that are defined in the class (or a subclass thereof) that they are being included into. Using `Comparable` as an example, it has an interface requiring the method <=>. Note that the same can apply to classes that have methods that reference a method that doesn't exist in the class; to be properly used, a subclass declaring that method must exist and that subclass must have been instantiated. This is similar behaviour to *abstract* classes in (for example) JAVA.

In a dynamically typed programming language, it is often found that inheritance is used predominantly for implementation inheritance. As inheritance (outside of reflection and meta-programming) has little effect on the runtime of a program, a subclass can (from an analysis point of view) be considered to be a copy of its superclass/modules with some methods added or overridden (subsection 2.5.2).

[6] Often called prototype programming - for example JAVASCRIPT, [Net].

[7] This is *single inheritance*. PYTHON is a dynamic Object-Oriented language with *multiple inheritance*.

2.5.2 Overriding

A class may define a method with the same name as one in one of its superclasses or ancestor modules. This is *overriding* (see Program 1). In some languages (e.g. JAVA), overriding has to preserve the subtyping principle because subclasses are implicit subtypes of their classes and can be used wherever a superclass/supertype is required. However in RUBY classes only inherit implementation and not type, which can lead to runtime errors (see Program 2) if subtyping is assumed.

Program 1 An example of subclassing and method overriding

```
 1 class Animal
 2   def sound
 3     return "undetermined sound"
 4   end
 5   def make_noise
 6     puts sound
 7   end
 8 end
 9 class Dog < Animal
10   def sound
11     return "bark!"
12   end
13 end
14 if __FILE__== $0
15   animal = Animal.new
16   animal.make_noise
17   animal = Dog.new
18   animal.make_noise
19 end

> Program output...
undetermined sound
bark!
```

There are several strategies to typing that can take place here. We can create a type system to associate type with class, and require subclassing to preserve subtyping. Or we can separate the subtype and subclass relationship and maintain that subclassing only inherits implementation. With pluggable type systems we can, of course, have both. For this project, however, the notion of inheritance is more of a complicating factor; and since there is no subtyping in a class hierarchy, adding it to my RUBY subset would add noise to the language.

2.5.3 Operators

Operators, and whether the redefinition of them is a good thing is often a source of heated discussion for programmers. In, for example, C++, operators need not be defined by the class(es) they relate together, which can make understanding what operators do, and the locating of the source code backing the operator difficult. In JAVA, operators cannot be redefined and have well defined

Program 2 An example of bad subclassing

```
 1 require 'Code/Animal'
 2 class Alien < Animal
 3   def sound loud
 4     loud ?  "WOOWOO":  "woowoo"
 5   end
 6 end
 7 if __FILE__== $0
 8   animal = Alien.new
 9   animal.make_noise
10 end

> Program output...

Code/Animal.rb:6:in `sound': wrong number of arguments (0 for 1) (ArgumentError)
from Code/Animal.rb:6:in `make_noise'
from Code/BadAnimal.rb:9
```

use cases. However, it means that verbose method names, and method syntax must be used in instances where a simple + between two objects would be closer to the problem domain.

Both PYTHON and RUBY allow operators to be redefined, however operators are treated as syntactic sugar for normal method calls. In PYTHON the operators are specified using pre-declared aliases. For example `__add__` is the method name for +. If a class defines this method, then they have + defined as-well. RUBY doesn't require the aliased method name, you can define a method entitled + (see Program 3).

From a typing point of view, this style of behaviour makes operators equivalent to methods, and no more difficult or easy to type, since the receiver of the method is known as is the argument.

2.5.4 Classes, Class Methods, Instance Methods and Metaclasses

Many languages have a notion of a *class method*. This is a method that is defined on a class, but does not require an instance of that class to call it, and therefore cannot access any instance data (since it doesn't exist). In the JAVA model, these are `static` methods.

In RUBY, however, there are only instance methods, and all methods are looked up from a class or module. Classes are objects too, they are instances of the class `Class`, so it would make sense that a class method is defined on their class (i.e. `Class`). However, this would mean all class methods would be shared by every class.

To solve this RUBY has the notion of meta-classes[8]. Every object instance in RUBY can have methods defined upon it that do not become visible to their class. When these per-instance methods are defined, the instance creates a virtual class and internally holds a reference to it. Methods are looked up first from the virtual class, then from the normal class hierarchy.

[8]For those from a SMALLTALK background, the notion of meta-class here is different to what would be expected, for more see [TH01]

Program 3 Operator overloading in RUBY

```
 1 class Glass
 2
 3   def initialize(contents = nil)
 4     @contents = contents
 5   end
 6
 7   def + contents
 8     if @contents
 9       puts "I'm full!  There's #{contents} on the floor!"
10       return self
11     else
12       return Glass.new(contents)
13     end
14   end
15
16   def to_s
17     if @contents
18       return "I'm a glass with #{@contents} in me"
19     else
20       return "I'm an empty glass"
21     end
22   end
23
24 end
25
26 glass = Glass.new
27 puts glass
28
29 # Call as an operator
30 glass_with_water = glass + "water"
31 puts glass_with_water
32
33 # Call as a method
34 glass_with_water.+("an elephant")
```

```
> Program output...
I'm an empty glass
I'm a glass with water in me
I'm full! There's an elephant on the floor!
```

This has an interesting side effect, in JAVA you can invoke a static method from an instance of a class or from the class directly. In RUBY you can only invoke a class method from the class name directly (See Program 4).

The ability of any instance to gain methods that are not explicitly defined in its class hierarchy or any imported modules (together known as a class's ancestors) could potentially mean every instance variable must be associated with a set of method names and this information has to be propagated around. There are also issues, if, for example, a method gives an argument it is passed a new method.

For my project it is likely that such extensibility would be too much to consider at the outset, and should be restricted from the initial subset that is looked at. However, time permitting, it would be quite exciting to explore techniques for handling such extensibility of objects (and previous work does suggest it is possible, e.g. [AGD05]).

2.5.5 Closures

The functional programming world has long known that the ability to pass around functions as first-class entities can provide highly expressive and concise solutions to many programming problems. For example the `inject` (or `fold`) function, can be used to neatly build many functions over enumerable items (e.g. summation, product over numeric lists).

The same benefits can be reaped in an Object-Oriented setting. In the same way that an anonymous function can be built in a functional setting (the `(\ x sum -> x + sum )` in Program 5), in RUBY an anonymous code block (closure) can instead be built (the `{ | x, sum | x + sum }` in Program 6).

Code blocks / closures in RUBY have the added advantage of capturing the local variables that are available when the closure is declared, and being able to modify and read them potentially long after the method they are declared in has stopped running. In this way they are similar to JAVA anonymous inner classes, although with those it is not possible to re-assign accessed variables (they must be declared `final`).

First class code blocks are not unique to RUBY, they were available in SMALLTALK. Because of this the STRONGTALK (see subsection 2.2.4) developers encountered them. As already mentioned, those developers found that first-class code blocks can be used to handle many program idioms in a way that is very possible to type. Because of their fundamental use in the RUBY libraries, and their general power, they should be part of my RUBY subset.

2.5.6 Instance Variables

Objects (in a non-functional sense) have state and methods that can alter/update that state. Instance variables are how the state is managed, and are thus fairly fundamental in imperative Object-Oriented languages.

Some languages (C++, JAVA etc) require all instance variables to be pre-declared. Others (e.g. PYTHON, JAVASCRIPT) require explicit access to instance variables through some `self` reference, but do not need them to be pre-declared before use. RUBY also requires explicit access to instance variables, but this is marked syntactically by the use of an `@` prefix on the name of the variable.

Program 4 Class (or per-instance) methods in Ruby

```
 1 class Hello
 2
 3   # Define a class method
 4   def Hello.say_hello
 5     puts "Hello from the class Hello"
 6   end
 7
 8   # Define an instance method
 9   def say_hi
10     puts "Hi from an instance of class Hello"
11   end
12
13 end
14
15 #Call the class method
16 Hello.say_hello
17
18 #Call the instance method
19 hello = Hello.new
20 hello.say_hi
21
22 #Any instance can have a method defined on it
23 def hello.say_yo
24   puts "Yo from just this instance of hello"
25 end
26
27 hello.say_yo
28 hello2 = Hello.new
29
30 puts "hello methods:  "+ hello.methods.grep(/say/).join(" ")
31 puts "hello2 methods:  "+ hello2.methods.grep(/say/).join(" ")
32 puts "class Hello methods:  "+ Hello.methods.grep(/say/).join("
")
33

> Program output...
Hello from the class Hello
Hi from an instance of class Hello
Yo from just this instance of hello
hello methods: say_hi say_yo
hello2 methods: say_hi
class Hello methods: say_hello
```

Program 5 Uses of the HASKELL `foldr` function to build a summation function

```
1 add_up ::  [Int] -> Int
2 add_up list = foldr (\x sum -> x + sum ) (0) list
```

Program 6 Uses of the RUBY `inject` function to build a summation

```
1 puts [1,2,3].inject { |x,sum|  x + sum }

> Program output...
6
```

Instance variables are also not visible outside of the instance that holds them, which means there is encapsulation at the instance level, as opposed to the class based and access modified encapsulation of some other languages (JAVA), or the open access of JAVASCRIPT.

This instance-level encapsulation of state should make typing efforts easier in this project, although it is somewhat complicated by the fact that superclasses, subclasses and mixed-in modules all share the same set of instance variables for any instance, and that instance variables can still be aliased by variables outside of the class / module. The fact that all instance variables are syntactically marked means the lack of pre-declaration of variables does not mean that statically all possible instance variables cannot be found.

2.5.7 nil

Many languages have a notion of an undefined variable. In RUBY this notion is marked by the singleton sentinel object `nil`. `nil` and its distant-cousin `false` are the only two objects that will fail a truth (`if`, `while`) test. `nil` is also the default value for instance variables if accessed before being assigned. This differs from say, JAVASCRIPT, where there is a notion of `null` and also of `undef`.

Many of the papers that deal with type inference or type checking of existing languages leave the checking of the `null` variable to data flow analysis; i.e that the type system will try to guarantee that either the method to be called is present (or the class of the object is what is expected), or that the object is `null/nil/undef` as appropriate.

However in the case of my project, it seems slightly counter-intuitive to make special cases for `nil`. Ideally the programmer would like to know if they are going to try and call methods on the `nil` object (which also does have some legitimate method calls that can be made on it). Simple type systems can be developed to ensure that `nil` variables are not passed into methods that expect non-`nil` objects, but this is just a subset of the passing-the-wrong "type" of argument into a method problem.

2.5.8 Libraries

RUBY has a large and comprehensive set of existing libraries. While it would be beyond the scope of the project to attempt to type or analyse / validate all of them (especially since many are implemented in C), the fact that my project is based upon local-code analysis only means that I should

be able to provide type declarations for a core of the library elements, for those type systems / checkers that could use them.

2.5.9 Summary

The subset of RUBY I have chosen is representative of a structural, dynamically checked, Object-Oriented programming language. There are many interesting typing problems to be tackled that can be incrementally developed using pluggable type systems. There are also extensions I can look at (for example inheritance or), time permitting. In chapter 3 I formally define the subset of RUBY this project will focus on.

2.6 Type Systems

2.6.1 Definition

From the literature, a definition of a type system compatible with what will be discussed in this report could be:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute. [Pie02, pg.1]

In the context of this project, a program is considered to be an instance of some defining grammar and the phrases are the sub-expressions found in that instance.

Alternatively, a type system can be seen as a function that takes a program and returns a boolean representing the acceptance or rejection of that program with respect to certain behaviours or properties the program may have.

2.6.2 Static and Dynamic Type Systems

These two terms refer to when type checking takes place. A statically typed programming language will enforce / perform type checking at a non-runtime time. (Usually referred to as compile time). Dynamic programming languages will check and enforce any type rules as they run.

2.6.3 Safe and Unsafe Type Systems

These two terms explain the guarantees the type system may have. An unsafely typed programming language will have ways to subvert or escape from the type system, creating non-deterministic or unspecified behaviour. A safely typed system will not allow such subversion to happen.

A good example in this case are casts in C compared to casts in JAVA. A JAVA cast is checked at runtime, keeping the system safe. A cast in C is (generally) not, so pointers can end up referring to entities not reflected by their known static type and causing non-deterministic, destructive behaviour.

2.6.4 Structural and Nominative Subtype Relationship

These two terms (and possibly others) describe how the subtype relationship is defined. Nominative typing is where there is a named hierarchy of types (usually this is backed by an implementation hierarchy of classes and subclasses). Structural typing is based upon the structure of types, for example the names and arities of methods an instance supports could be its structure.

2.6.5 Manifest Typing and Type Inference

These terms describe how the types are declared or expressed. In a manifest type system, the program must have type annotations provided for it by the developer. These are then (hopefully) validated by the type checker. Alternatively a programming language can employ type inference to ascertain the type annotations automatically before or during verification.

2.6.6 Optional, Mandatory and No Typing

These terms describe whether there is a type system. A mandatory type system is one that must be encountered, an optional one may be encountered, and no typing implies that there is no type system. The untyped λ-calculus is an example of a no-typing system.

2.6.7 Soundness and Completeness

Traditionally, type systems for statically typed programming languages are designed to be (hopefully) provably *sound*. A sound type system checking for an undesirable property will only accept programs without that property. It will therefore always reject programs with the undesirable property. However it may reject programs without the property as it is unable to prove the absence of it. Sound type systems err on the side of caution.

For statically typed languages this is important as there may be no way to do further checking at run-time (or perhaps a guarantee of run-time freedom from error is important). In this way accepted programs are *always safe*, and rejected programs *may be unsafe*.

Alternatively, type systems can be designed to be *complete*. If a program is free of some undesirable property, then a complete type system checking for that property will always accept that program. However a complete type system may accept programs with the undesirable property. In this way rejected programs are *always unsafe*, and accepted programs *may be safe*.

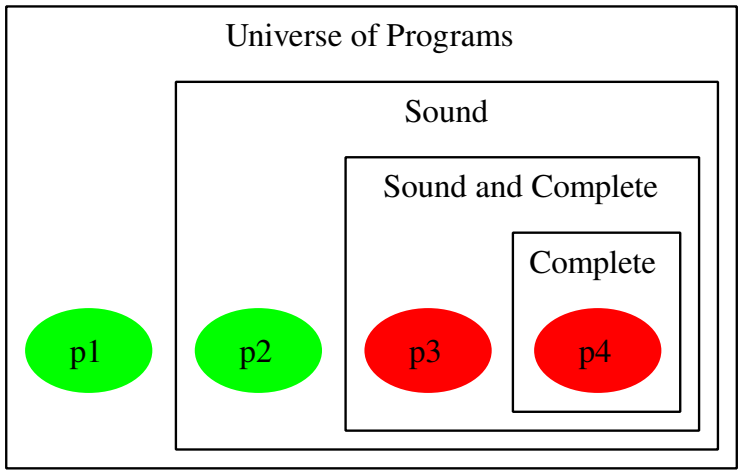
Programming languages that utilise complete type systems need to be coupled with some form of run-time checking in order to ensure the state of the run-time-system remains safe with respect to the undesirable property the type system is checking for.

A type system that is both *sound and complete* will be able to perfectly classify programs, rejecting only programs with the undesirable property and accepting only programs without it.

2.7 Conclusion

In this section, the relevant background literature and theory for this project has been discussed. In order to demonstrate the idea of pluggable type systems, an example programming language

Example 1 Sound and complete type systems. The red programs (p3 and p4) have an undesirable property, the green programs (p1 and p2) are free of it. The type systems will reject programs contained within them, and accept those outside.



has been developed, using some of the features that were mentioned in this chapter. The example programming language is the subject of the next chapter.

Chapter 3

The Language $\mathcal{R}_{sub}$

3.1 Introduction

In this chapter I present a simple, dynamically typed programming language entitled $\mathcal{R}_{sub}$. The structure and syntax of the language is fully defined, before the operational semantics and runtime well formed rules are given.

A formal presentation is given so that it could be possible to reason about the type systems that will be developed later. Most modern dynamically typed programming languages are specified by their implementation, which can make rigorously exploring their behaviour difficult on a case by case basis.

$\mathcal{R}_{sub}$ is (with one known exception[1]) a restricted subset of the programming language RUBY. RUBY has been chosen as it is a modern, dynamically typed, object-oriented programming language that is very powerful and clean.

$\mathcal{R}_{sub}$models the class-based object-oriented design of RUBY, but to simplify the presentation, inheritance has been removed[2]. As explained in subsection 2.5.1 the use of inheritance here is generally to reuse implementation instead of polymorphism (since classes do not necessarily represent a type). $\mathcal{R}_{sub}$ does not feature any control flow primitives other than method call, and instead uses closures[3] to emulate them.

3.2 The structure and syntax of $\mathcal{R}_{sub}$

$Program = ClassId \rightarrow (\ MethodId \rightarrow meth)$

where

[1] $\mathcal{R}_{sub}$ Closure arguments will shadow a similarly named local variable in the enclosing scope, RUBY closure arguments with a similar name will just refer to the original variable. Since local variables can simply be renamed to avoid this problem, it shall not be considered further.

[2] For the interested, the original $\mathcal{R}_{sub}$ formulation had multiple inheritance, and only a few small changes are needed to go from one to the other.

[3] Commonly called *blocks* in RUBY. See subsection 2.5.5

e	::=		*expressions from the set Expr*
	|	lvar l	*Local variable lookup*
	|	ivar i	*Instance variable lookup*
	|	$e.m(\ e)$	*Method call*
	|	new c	*New instance creation*
	|	lvar $l = e$	*Local variable assignment*
	|	ivar $i = e$	*Instance variable assignment*
	|	$\{\|l\|\ e\ \}$	*Closure creation*
	|	nil	*Literal* nil *object*
	|	self	*Self lookup*
	|	$e;\ e$	*Sequence*
$meth$	::=	$\|l\|\ e$	*Method definition*

With the identifier conventions

c	$\in$	$ClassId$
m	$\in$	$MethodId$
l	$\in$	$LVarId$
i	$\in$	$IVarId$

3.2.1 Well Formed Programs

$$\frac{\vdash\ P\ \diamond_{nil}}{\vdash\ P\ \diamond}$$ Well formed programs

$$\frac{\begin{array}{l} NilClass\ \in\ \mathrm{dom}(P) \\ P(NilClass) = (\ \emptyset) \end{array}}{\vdash\ P\ \diamond_{nil}}$$ NilClass

3.3 The operational semantics of $\mathcal{R}_{sub}$

Definitions

The operational semantics are defined by a re-writing operation, that re-writes expressions, stacks and memories into results, new stacks and new memories.

$\rightsquigarrow$: $Program \rightarrow e \times stack \times memory \rightarrow result \times stack \times memory$

where

$stack$	=	$(\ LVarId \rightarrow ref\ ) \times addr$	(*local variables references*, *location of* self)
$heap$	=	$(\ addr \rightarrow val)$	
$memory$	=	$heap \times (\ ref \rightarrow addr) \times (\ \{\ \mathrm{nil}\ \} \rightarrow addr)$	(*heap*, *where lvars are in heap*, *where* nil *is in he*
val	=	$object \cup closure$	
$addr$	=	$\mathbb{N}^+$	
ref	=	$\mathbb{N}^+$	
$object$	=	$ClassId \times (\ IVarId \rightarrow addr)$	(*object's class*, *values for ivars*)
$closure$	=	$(\ \{\|l\|\ e\ \}) \times stack$	((*the closure*), *captured stack*)
$deviation$	=	$\{\ \mathrm{noMethodError},\ \mathrm{noLocalVariableError}\ \}$	
$result$	=	$deviation \cup addr$	

Symbol Convention

In the rules for the operational semantics, the following symbol convention is used:

$\sigma \in stack$
$\chi \in heap$
$\mathcal{M} \in memory$
$\iota \in addr$
$\rho \in ref$
$o \in object$
$\varsigma \in closure$
$vl \in val$
$re \in result$
$dv \in deviation$

3.3.1 Some utility functions

Stack implicit lookup

$$\sigma(l) \equiv \sigma\downarrow_1(l)$$

Stack update

$\sigma[\, l \mapsto \rho]$ *gives a stack s.t.*
$\sigma[\, l \mapsto \rho]\downarrow_1(l) \equiv \rho$
$\sigma[\, l \mapsto \rho]\downarrow_1(l') \equiv \sigma\downarrow_1(l')$ *where* $l \not\equiv l'$
$\sigma[\, l \mapsto \rho]\downarrow_2 \equiv \sigma\downarrow_2$

Memory implicit lookup

$$\mathcal{M}(\iota) \equiv \mathcal{M}\downarrow_1(\iota)$$

Memory implicit update

$\mathcal{M}[\, \iota \mapsto vl]$ *gives a memory s.t.*
$\mathcal{M}[\, \iota \mapsto vl]\downarrow_1(\iota) \equiv vl$
$\mathcal{M}[\, \iota \mapsto vl]\downarrow_1(\iota') \equiv \mathcal{M}\downarrow_1(\iota')$ *where* $\iota \not\equiv \iota'$
$\mathcal{M}[\, \iota \mapsto vl]\downarrow_2 \equiv \mathcal{M}\downarrow_2$
$\mathcal{M}[\, \iota \mapsto vl]\downarrow_3 \equiv \mathcal{M}\downarrow_3$

Memory implicit update 2

$\mathcal{M}[\, \rho \mapsto \iota]_2$ *gives a memory s.t.*
$\mathcal{M}[\, \rho \mapsto \iota]_2\downarrow_1 \equiv \mathcal{M}\downarrow_1$
$\mathcal{M}[\, \rho \mapsto \iota]_2\downarrow_2(\rho) \equiv \iota$
$\mathcal{M}[\, \rho \mapsto \iota]_2\downarrow_2(\rho') \equiv \mathcal{M}\downarrow_2(\rho')$ *where* $\rho \not\equiv \rho'$
$\mathcal{M}[\, \rho \mapsto \iota]_2\downarrow_3 \equiv \mathcal{M}\downarrow_3$

Method lookup

$$\begin{array}{ll} \mathit{Method}(c,\ m) = |l|\ e & \mathit{where}\ P(c)(m) = |l|\ e \\ \mathit{Method}(c,\ m) = \ \mathit{Undef} & \mathit{otherwise} \end{array}$$

3.3.2 The Operational Semantics rules

Standard Cases

$$\frac{\iota = \ \mathcal{M}\downarrow_2(\sigma(\ l))}{\text{lvar}\ l,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}}$$ Local Variable

$$\frac{\sigma(l) = \ \mathit{Undef}}{\text{lvar}\ l,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \text{noLocalVariableError},\ \sigma,\ \mathcal{M}}$$ Local Variable (no variable error)

$$\frac{\iota = \ \mathcal{M}(\sigma\downarrow_2)\downarrow_2(i)}{\text{ivar}\ i,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}}$$ Instance Variable

$$\frac{\begin{array}{l}\mathcal{M}(\sigma\downarrow_2)\downarrow_2(i) = \ \mathit{Undef} \\ \text{ivar}\ i = \ \text{nil},\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}'\end{array}}{\text{ivar}\ i,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}'}$$ Instance Variable (uninited access)

$$\frac{\begin{array}{l} e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota_1,\ \sigma_1,\ \mathcal{M}_1 \\ e_2,\ \sigma_1,\ \mathcal{M}_1 \rightsquigarrow_P\ \iota_2,\ \sigma',\ \mathcal{M}_2 \\ \mathcal{M}_2(\iota_1) = (\ c,\ \_) \\ \mathit{Method}(c,m) = |l|\ e_3 \\ \mathcal{M}_3 = \ \mathcal{M}_2[\ \rho \mapsto\ \iota_2]_2,\ \rho \notin\ \text{dom}(\mathcal{M}_2\downarrow_2) \\ \sigma_2 = (l \mapsto\ \rho,\ \iota_1) \\ e_3,\ \sigma_2,\ \mathcal{M}_3 \rightsquigarrow_P\ re,\ \_,\ \mathcal{M}' \end{array}}{e_1.m(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ re,\ \sigma',\ \mathcal{M}'}$$ Method Call (normal)

$$\frac{\begin{array}{l} e_1,\ \sigma,\ \chi \rightsquigarrow_P\ \iota_1,\ \sigma_1,\ \mathcal{M}_1 \\ e_2,\ \sigma_1,\ \chi_1 \rightsquigarrow_P\ \iota_2,\ \sigma',\ \mathcal{M}_2 \\ \mathcal{M}_2(\iota_1) = (\{|l|\ e_3\},\ \sigma_2) \\ \mathcal{M}_3 = \ \mathcal{M}_2[\ \rho \mapsto\ \iota_2]_2,\ \rho \notin\ \text{dom}(\mathcal{M}_2\downarrow_2) \\ \sigma_3 = \ \sigma_2[\ l \mapsto\ \rho] \\ e_3,\ \sigma_3,\ \mathcal{M}_3 \rightsquigarrow_P\ re,\ \_,\ \mathcal{M}' \end{array}}{e_1.\mathit{call}(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ re,\ \sigma',\ \mathcal{M}'}$$ Method Call (stack change)

$$\frac{\begin{array}{l} e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota_1,\ \sigma_1,\ \mathcal{M}_1 \\ e_2,\ \sigma_1,\ \mathcal{M}_1 \rightsquigarrow_P\ \_,\ \sigma',\ \mathcal{M}' \\ \mathcal{M}'(\iota_1) = (\ c,\ \_) \\ \mathit{Method}(c,m) = \ \mathit{Undef} \end{array}}{e_1.m(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \text{noMethodError},\ \sigma',\ \mathcal{M}'}$$ Method Call (no method error normal)

$$\frac{\begin{array}{l} e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota_1,\ \sigma_1,\ \mathcal{M}_1 \\ e_2,\ \sigma_1,\ \mathcal{M}_1 \rightsquigarrow_P\ \_,\ \sigma',\ \mathcal{M}' \\ \mathcal{M}'(\iota_1) = \varsigma \\ m \neq\ call \end{array}}{e_1.m(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \text{noMethodError},\ \sigma',\ \mathcal{M}'}\ \text{Method Call (no method error closure)}$$

$$\frac{\begin{array}{l} o = (\ c, \emptyset) \\ \mathcal{M}' =\ \mathcal{M}[\iota \mapsto\ o],\ \iota \notin\ \text{dom}(\mathcal{M}\downarrow_1) \end{array}}{\text{new}\ c,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}'}\ \text{New}$$

$$\frac{\begin{array}{l} e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma',\ \mathcal{M}_1 \\ \sigma'(l) =\ \rho \\ \mathcal{M}' =\ \mathcal{M}_1[\rho \mapsto\ \iota]_2 \end{array}}{\text{lvar}\ l =\ e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma',\ \mathcal{M}'}\ \text{Local Assignment (update)}$$

$$\frac{\begin{array}{l} e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma_1,\ \mathcal{M}_1 \\ \sigma' =\ \sigma_1[\ l \mapsto\ \rho],\ l \notin\ \text{dom}(\sigma_1\downarrow_1) \wedge\ \rho \notin\ \text{dom}(\mathcal{M}_1\downarrow_2) \\ \mathcal{M}' =\ \mathcal{M}_1[\rho \mapsto\ \iota]_2 \end{array}}{\text{lvar}\ l =\ e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma',\ \mathcal{M}'}\ \text{Local Assignment (create)}$$

$$\frac{\begin{array}{l} e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma',\ \mathcal{M}_1 \\ (\ c,\ f_{iv}) =\ \mathcal{M}_1(\sigma'\downarrow_2) \\ \mathcal{M}' =\ \mathcal{M}_1[\sigma'\downarrow_2 \mapsto\ (c,\ f_{iv}[i \mapsto\ \iota])] \end{array}}{\text{ivar}\ i =\ e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma',\ \mathcal{M}'}\ \text{Instance Assignment}$$

$$\frac{\begin{array}{l} \varsigma = (\{|l|\ e\},\ \sigma) \\ \mathcal{M}' =\ \mathcal{M}[\iota \mapsto\ \varsigma],\ \iota \notin\ \text{dom}(\mathcal{M}\downarrow_1) \end{array}}{\{|l|\ e\},\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}'}\ \text{Closure Creation}$$

$$\frac{\iota =\ \mathcal{M}\downarrow_3(\text{nil})}{\text{nil},\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}}\ \text{nil}$$

$$\frac{\iota =\ \sigma\downarrow_2}{\text{self},\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota,\ \sigma,\ \mathcal{M}}\ \text{self}$$

$$\frac{\begin{array}{l} e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota',\ \sigma_1,\ \mathcal{M}_1 \\ e_2,\ \sigma_1,\ \mathcal{M}_1 \rightsquigarrow_P\ re,\ \sigma',\ \mathcal{M}' \end{array}}{e_1;\ e_2,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ re,\ \sigma',\ \mathcal{M}'}\ \text{Sequence}$$

Error propagation

$$\frac{e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma_1,\ \mathcal{M}_1}{e_1;\ e_2,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv}$$ Sequence Error Propogation Left

$$\frac{e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}{e_1.m(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}$$ Method Call Error Propogation Left

$$\frac{\begin{array}{l}e_1,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ \iota_1,\ \sigma_1,\ \mathcal{M}_1\\ e_2,\ \sigma_1,\ \mathcal{M}_1 \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'\end{array}}{e_1.m(e_2),\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}$$ Method Call Error Propogation Arg

$$\frac{e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}{\text{lvar } l =\ e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}$$ Local Assignment Error Propogation

$$\frac{e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}{\text{ivar } i =\ e,\ \sigma,\ \mathcal{M} \rightsquigarrow_P\ dv,\ \sigma',\ \mathcal{M}'}$$ Instance Assignment Error Propogation

3.3.3 Well formed memories and stacks

$$\frac{\begin{array}{l}\vdash\ \mathcal{M} \diamond_{lv}\\ \vdash\ \mathcal{M} \diamond_{stack}\\ \vdash\ \mathcal{M} \diamond_{nil}\end{array}}{\vdash\ \mathcal{M} \diamond}$$ Well formed Memories

$$\frac{\begin{array}{l}\mathcal{M} = (\ \chi,\ f_{lv},\ \_)\\ \forall\, \iota\ \in\ \text{range}(f_{lv}) :\ \iota\ \in\ \text{dom}(\chi)\end{array}}{\vdash\ \mathcal{M} \diamond_{lv}}$$ Captured local variables

$$\frac{\begin{array}{l}\forall(\varsigma,\ \sigma)\ \in\ \text{range}(\mathcal{M}\downarrow_1)\\ \mathcal{M} \vdash\ \sigma \diamond\end{array}}{\vdash\ \mathcal{M} \diamond_{stack}}$$ Captured stacks

$$\frac{\begin{array}{l}\mathcal{M} = (\ \chi,\ \_,\ f_{nil})\\ f_{nil}(\text{nil}) =\ \iota\\ \chi(\iota) = (\ \mathit{NilClass},\ \emptyset)\end{array}}{\vdash\ \mathcal{M} \diamond_{nil}}$$ Nil

$$\frac{\begin{array}{l}\sigma = (\ f_{lv},\ \iota)\\ \mathcal{M}\downarrow_1(\iota) =\ o\\ \forall\, \rho\ \in\ \text{range}(f_{lv}) :\ \rho\ \in\ \text{dom}(\mathcal{M}\downarrow_2)\end{array}}{\mathcal{M} \vdash\ \sigma \diamond}$$ Well formed stack

3.4 Some examples of $\mathcal{R}_{sub}$ behaviour

To demonstrate the interactions of closures with local variables and their enclosing scopes, I now give a few sample programs and the results of their execution. All of these example programs were written using the editor developed in this project, and the results of any executions are captured outputs from real executions of those programs. The formatting of the code of the examples was done using a tree walker that converted the abstract-syntax-tree representation of the program code to latex.

Program 7 $\mathcal{R}_{sub}$ closures are able to update the values of local variables when called

```
 0:  lvar x = 7;
 1:  lvar b = {|| lvar x = 6 };
 2:  lvar b.call();
 3:  print(lvar x);

> Program output...
6
```

Program 8 Closures passed to other methods are able to alter the local variables from the scope they came, regardless of the scope they are executed in

```
 0:  ExampleClass ↦
 1:    block_calling_method ↦ |b|
 2:      lvar b.call();

 3:    example1 ↦ ||
 4:      lvar x = 7;
 5:      lvar b = {|| lvar x = 6 };
 6:      self.block_calling_method(lvar b);
 7:      print(lvar x);

 8:    example2 ↦ ||
 9:      lvar b = {|| lvar x = 6 };
10:      self.block_calling_method(lvar b);
11:      print(lvar x);

12:  new ExampleClass.example1();
13:  new ExampleClass.example2();

> Program output...
6
NameError: x does not exist (RuntimeError)
```

Program 9 New local variables created in $\mathcal{R}_{sub}$ closures are not visible outside the scope of the closure

```
 0:  lvar b = {|| lvar x = 6 };
 1:  lvar b.call();
 2:  print(lvar x);

> Program output...

NameError: x does not exist (RuntimeError)
```

Program 10 Closures received from other methods do not alter the local variables in the scope they are executed in if it is not the scope they were created in

```
 0:  ExampleClass ↦
 1:    make_a_block ↦ ||
 2:      {|| lvar x = 6 };

 3:    example1 ↦ ||
 4:      lvar x = 3;
 5:      lvar b = self.make_a_block();
 6:      lvar b.call();
 7:      print(lvar x);

 8:    example2 ↦ ||
 9:      lvar b = self.make_a_block();
10:      lvar b.call();
11:      print(lvar x);

12:  new ExampleClass.example1();
13:  new ExampleClass.example2();

> Program output...
3
NameError: x does not exist (RuntimeError)
```

Program 11 If an $\mathcal{R}_{sub}$ closure creates a new local variable, then it cannot change the value of an equivalently named local variable created after the closure outside its scope

```
 0:  lvar b = {|| lvar x = 6 };
 1:  lvar x = 3;
 2:  lvar b.call();
 3:  print(lvar x);

> Program output...
3
```

Program 12 Closures that declare new local variables have their own copies of them

0: lvar $b = \{||\,\text{lvar}\ y = 0; \{||\,\text{lvar}\ y = \text{lvar}\ y. + (1); \text{lvar}\ y;\ \};\ \};$
1: lvar *counter1* = lvar $b.call();$
2: lvar *counter2* = lvar $b.call();$
3: $print($lvar *counter1*$.call());$
4: $print($lvar *counter2*$.call());$
5: $print($lvar *counter1*$.call());$
6: $print($lvar *counter1*$.call());$
7: $print($lvar *counter2*$.call());$
8: $print($lvar *counter2*$.call());$

```
> Program output...
112323
```

3.5 Control Flow and Numerals

The following $\mathcal{R}_{sub}$ source code demonstrates how conditional statements, while loops and simple numerals can be encoded. The inspiration for some of the encoding comes from the SMALLTALK language, which also does not provide syntax for conditional statements or loops[GR83]. The use of *ifTrue* and *ifFalse* methods that take closures and execute them depending on whether a `True` or `False` object receives them being the main example. As closures are unable to respond to any message other than *call*, the *whileTrue* message is part of a special `While` class which uses *whileTrue* for initialization and *do* to perform the loop.

```
0:   True ↦
1:    ifTrue ↦ |b|
2:     lvar b.call();

3:    ifFalse ↦ |b|
4:     self;

5:    not ↦ ||
6:     new False;


7:    False ↦
8:     ifTrue ↦ |b|
9:      self;

10:    ifFalse ↦ |b|
11:     lvar b.call();

12:    not ↦ ||
13:     new True;


14:   While ↦
15:    whileTrue ↦ |b|
16:     ivar cond = lvar b;
17:     self;

18:    do ↦ |b|
19:     ivar cond.call().ifTrue({|| lvar b.call(); self.do(lvar b); });
```

```
20:  Zero ↦
21:    zero? ↦ ||
22:      new True;

23:    equals? ↦ |other|
24:      lvar other.zero?();

25:    succ ↦ ||
26:      new Succ.ctor(self);

27:    times ↦ |b|
28:      nil;


29:  Succ ↦
30:    ctor ↦ |pred|
31:      ivar pred = lvar pred;
32:      self;

33:    zero? ↦ ||
34:      new False;

35:    equals? ↦ |other|
36:      lvar ret = new False;
37:      lvar other.zero?().ifFalse({|| lvar ret = ivar pred.equals?(lvar other.pred()) });
38:      lvar ret;

39:    succ ↦ ||
40:      new Succ.ctor(self);

41:    times ↦ |b|
42:      lvar b.call(self);
43:      ivar pred.times(lvar b);

44:    pred ↦ ||
45:      ivar pred;


46:  new Succ.ctor(new Succ.ctor(new Zero)).times({|num| print('!') });
47:  lvar x = new Succ.ctor(new Zero).succ();
48:  new While.whileTrue({|| lvar x.zero?().not() }).do({|| lvar x = lvar x.pred(); print('*'); });
```

```
> Program output...
!!**
```

3.6 Conclusion

In this chapter the programming language $\mathcal{R}_{sub}$ has been formalised, and some examples of the results of its execution given. This language is used in the following chapter to give a concrete example of how pluggable type systems could work.

Chapter 4

Type Systems for $\mathcal{R}_{sub}$

4.1 Introduction

In the previous chapter a simple dynamically typed programming language, entitle $\mathcal{R}_{sub}$ developed. The focus of this chapter is to give an example of a pluggable type system for that language. The different parts of such a system (the programming language, type annotations, type annotators, type checkers) will be introduced, and a demonstration of how they can interact in a useful way.

4.2 Overview

The overall aim is to associate meta-data (annotations) to points on the abstract-syntax-tree of the program. This meta data can be inferred by the system (annotators) or placed there by the user. For the purposes of this project, the human user can be thought of as a special type of annotator. These annotations are then checked by type checkers to ensure certain program properties hold.

The annotators are able to both look at the syntax of the program, and also see what other annotations are currently present on the tree.

4.3 An Example

Consider Program 13 (pg 36). Statically there is a lot of information that can be derived about this program in a fairly simple way, which will now be explained. This example will discuss the creation of a pluggable type system by incrementally looking for more properties, and creating new type checkers at appropriate times.

4.3.1 Annotator: seeding known values

On lines 2 and 3 we see the creation of two new objects, an instance of class `B` (`new B`) and a closure (`{ || ...}`). Since we know that those two sub-expressions always create their respective object, we can annotate them to that effect. For this we can create an annotation of the form `hasValue:[className|closureRef]`. So the `new B` of line 2 is annotated with

Program 13 A simple example program

```
0:   A ↦
1:    meth1 ↦ |arg|
2:     lvar x = new B;
3:     lvar y = {|| lvar arg.bark(lvar x) };
4:     lvar x.hello();
5:     lvar y.call();
6:     lvar x.goodbye();
7:     lvar arg.quack(lvar x);
8:     lvar y.call();
9:     lvar x.callclosure(lvar y);

10:  B ↦
11:   hello ↦ ||

12:   callclosure ↦ |cls|
13:    lvar cls.call();
```

`hasValue:[class B]`. We have designed our first annotator, which looks for new object or closure creations and annotates them with their class name or a closure reference.

4.3.2 Annotator: Final local variables

Focusing on `meth1`, all local variables within the method are only initialised once. For the method argument this is at calling time, and for the other local variables (`x, y`) when they are initially assigned (lines 2, 3). This information we will use for further analysis so we create a new annotation of the form `finalLVar:[lvarName]`, and annotate the method with annotations for each final variable; `finalLVar:[arg]`, `finalLVar:[x]`, `finalLVar:[y]`. If closures were to create their own local variables, we would annotate the closure with the final-information for those locals.

4.3.3 Annotator: Propagating known values to final local variables

The `hasValue:[]` annotation attached to a sub-expression expresses that the sub-expression will evaluate to either an object of the given class or the given closure. If such an expression is assigned to a `finalLVar:[]` local variable, then all uses of that local variable will evaluate to the same object. We can then copy the `hasValue:[]` annotation to all sub expressions that represent a use of that local variable. In our code listing, for example, all sub expressions of the form *lvar x* will be annotated with `hasValue:[class B]`.

4.3.4 Type Checker: Checking method calls on known values

There is enough information in the system to start type checking some method calls. If the sub expression for the receiver of the method call is annotated with `hasValue:[class someClass]`,

then a check can be made to see if `someClass` supports the method to be invoked. Similarly, a `hasValue:[someClosure]` will be checked that the method is named *call*. Note that a lack of a `hasValue:[]` annotation on the receiver does not mean we raise an error, it just means we don't have enough information to statically say an error will occur. The focus of these initial type systems is to highlight definite errors, and not places where there is not enough information to be able to decide.

4.3.5 Annotator: Method argument interface

With the above system we now find that method calls made on final local variables with known values are checked. It has been assumed, however, that the declaration of the variable will generally be correct, but the usage of the variable may be incorrect; errors are raised on invocations and the declaration class/closure define constraints.

With the argument to a method the situation is generally reversed. It is unknown what class/-closure the argument value will take; however the interface/protocol for the argument will be defined by the methods that are invoked upon it. Method calls invoked directly upon a method argument labelled `finalLVar:[]` will generally need to supported by the argument. However closures complicate this matter slightly; for example the execution of a closure may depend upon some condition that indicates the argument supports a method the closure makes use of. A new annotation to represent the interface expected of the method argument is created (`argMethod:[name]`), and for every method invocation (outside of a closure) made upon a `finalLVar:[]` that is a method argument we add to the method declaration a `argMethod:[methodName]` annotation. For example, method *meth1* will have the following annotation added; `argMethod:[quack]` (from line 7).

We also know that if the closure defined on line 3 is executed, then the method argument should support the method *bark*. To not waste this information, we can annotate the closure with an `argMethod:[bark]` annotation which can be reused later.

4.3.6 Annotator: Always Executed Closures

Knowing whether the closure defined on line 3 is always going to be executed would allow the `argMethod:[bark]` annotation to be propagated up to the method definition. One simple way initially of ascertaining this is to look for method calls upon sub expressions annotated with `hasValue:[closure]` and method name *call*. Initially we only look for these outside of closures. The definitely executed closures found can then be annotated with `alwaysExecuted:[]`. An `alwaysExecuted:[]` annotated closure can also be searched for other closures that are always executed, and they can also be annotated.

4.3.7 Annotator: Propagating known argument methods

Knowing which closures are definitely executed means the `argMethod:[]` annotations placed on those closures can be propagated up to the method to be added to the interface expected of the methods argument.

4.3.8 Checker: Correct Method Arguments

Method calls made where both the receiver and the argument have `hasValue:[]` annotations can be checked to ensure that the argument supports all the `argMethod:[]` annotations placed on the appropriate method of the receiver.

4.3.9 Other Possibilities

There are many ways in which the above system could be improved with more or smarter annotators or checkers. Two immediate examples could be, a non final local variable that is always defined to something of the same sub expression value could have a `hasValue:[]` annotation propagated onto it. Also keeping track of any aliasing of the method argument may allow other `argMethod:[]` annotations to be inferred.

4.4 Programmer Annotating

The annotations and type checking systems presented thus far are designed to supplement, and not replace the working practices of dynamically-typed programmers. The emphasis has been on type (or annotation) inference, since it involves no extra work for the programmer. However allowing programmers to add their own annotations can serve several purposes.

Since the annotations are there to represent a property, it is possible to allow a programmer to add their own annotations if they know a property holds which the system can't infer yet. For example, a `hasValue:[]` annotation for some reflexive code or `alwaysExecuted:[]` annotations for closures they know will be executed but the system can't prove. Ideally it would be in the programmer's interest to develop an annotator to place these annotations automatically for their specific case.

There is also the case that the programmer may want to make an assertion that they want the system to check is maintained. Annotations could be reused for this purpose. For example, they may want to explicitly express the property that a given method argument will always be final, and create an annotation to represent this. A type checker can then be developed to ensure no assignments are placed upon it. Alternatively a company may enforce a coding standard that means all method arguments are final, and a type checker can be developed for this too.

4.5 Conclusion

In this chapter, an example pluggable type system has been given. It features annotators and type checkers. Some details on the implications and possible issues of allowing programmers to annotate code have also been discussed. In the following chapter (chapter 5) the process of type annotating and checking is given more formally; this chapter gives a grounded example of the core of that process.

Chapter 5

Type Checking Algorithm

5.1 Introduction

In this chapter, the concepts used in chapter 4 are abstracted and formalised to give a framework in which type checking can take place. A simple algorithm is then given for how type checking in a pluggable environment could take place. With the algorithm in place, there are constraints that need to be met to ensure it terminates and these are discussed. Finally, the changes needed for iterative type checking within the given framework are discussed and the potential implications thereof.

5.2 Concepts

5.2.1 Defining Grammars and Programs

In the framework to be described there is a notion of a *Defining Grammar* and a *Program*.

Traditionally, the syntactic structure of a language is defined using some variation of Backus-Naur Form (BNF). This is the language's grammar and any fully complete instance of this grammar is a program.

A feature of most dynamically typed programming languages is that every complete instance of their defining grammar (i.e. every expressible program) is executable. The program may instantly fail with an error when executed, but it can be executed and there are semantics associated with it.

Within the framework to be defined, we need to take a slightly altered view of defining grammars and programs. Here, grammars need to be annotated to give names to the non-terminals in the production rules, and also to the individual production rules. (See Example 2).

An instance of this annotated grammar (a program) can then be viewed as a tree. The nodes of the tree take the names of the production rules and children of a node are named as per the annotations on the BNF.

Notation

$\mathcal{G}$ will be a defining grammar, Π will denote the set of all programs that are instances of $\mathcal{G}$, and $\pi \in \Pi$ is a program. `SExp` is the set of all sub-expressions in a program, and $\mathcal{S} \in$ `SExp` is an

Example 2 Example of an annotated BNF-like Extract from $\mathcal{R}_{sub}$

```
e ::=          : expressions from the set Expr
  | [mcall] e[lhs].m[name]( e[arg] ) : Method call
  | [new] new c[name]                : New instance creation
  | [closure] { |l[var]| e[code] }   : Closure creation
  | [nil] nil                        : Literal nil object
  | [self] self                      : Self lookup
  | [seq] e[lhs]; e[rhs]             : Sequence
  ...
```

individual sub-expression. A sub-expression of a program is any part of the abstract syntax tree of a program instance, not just the expressions that make up a method body.

5.2.2 Annotations, Annotators and Environments

Annotations are meta-data associated with specific nodes on the abstract syntax tree the of a given program. They represent the presence of a particular property of the tree, and can therefore be thought of as types. The absence of an annotation however means that it is unknown whether the property holds, not that the property does not hold.

Annotations are placed on the abstract syntax tree by annotators. Annotators can use the presence of other annotations and the syntactic structure of the tree to place further annotations. A consequence of annotators being able to use existing annotations to place further annotations is that they may need to be re-executed in the presence of other annotators for all the possible inferable annotations to be placed on the tree.

Annotations can potentially subsume other annotations[1], if the property they express is implied by the property of another annotation. To model this a partial ordering between annotations is also necessary. A possible example of subsumption that is applicable to the dynamically typed, structural languages under consideration could be an annotation specifying a closure is executed at least once being subsumed by one specifying a closure is executed exactly twice.

The mapping of sub expressions on the abstract syntax tree to annotations is held by environments. These are simple functions.

Notation

$\mathcal{A}$ will be the set of all annotators, and $\alpha \in \mathcal{A}$ is an annotator. There is also a (finite) set of all annotations, Ψ and $\psi \in \Psi$ is an annotation. The partial ordering between annotations, $\leq$ is defined such that if $\psi \leq \psi'$, then any property ψ guarantees, ψ' must also guarantee. In the simplest cases, $\leq$ can be taken to be the identity function.

We also need a mapping from sub-expressions to sets of annotations. This will be a function of the form Γ : `SExp` $\rightarrow \mathcal{P}(\Psi)$. With environments we can define annotators as functions that take a program and an environment and produce a new environment: $\alpha : \mathcal{P}(\Gamma) \times \Pi \rightarrow \mathcal{P}(\Gamma)$.

[1] My thanks to my second marker, Dr. Sophia Drossopoulou for pointing out the possibility of allowing this.

5.2.3 Type Checkers

Once all inferable annotations have been placed upon the tree by the annotators, the program will need to be type checked. This is done by specific type checkers, these use the annotations and the program syntax and either accept or reject the program.

Notation

Ω will be the set of all type checkers, and

$\omega \in \Omega$ will be an individual type checker. Type checkers are functions of the form $\omega : \Pi \times \mathcal{P}(\Gamma) \to \{\texttt{true}, \texttt{false}\}$.

5.3 Framework

5.3.1 Definition

With the preceding definitions in place, we can now define a framework, $\mathcal{F} = (\mathcal{A}, \Psi, \pi, \Omega)$.

5.3.2 Simple Type Checking Algorithm

With a framework instance, it is possible to fully annotate its program and then allow the type checkers to ascertain if the program is acceptable. An initial algorithm for this is given in Algorithm 1.

Algorithm 1: Simple Type Annotation and Checking algorithm for a Framework

Data: A framework; $\mathcal{F} = (\mathcal{A}, \Psi, \pi, \Omega)$
Result: A boolean

```
Γ ← (λx.∅);
repeat
    Γ′ ← Γ;
    forall α ∈ A do
        Γ ← α(Γ, π);
    end
until Γ = Γ′ ;
forall ω ∈ Ω do
    if not ω(π, Γ) then
        return false;
    end
end
return true;
```

This algorithm proceeds simply to initialize an empty environment, and then continuously runs the annotators over it until the environment ceases to change. It then allows the type checkers to run with resulting environment, and if any reject the program with that environment, the algorithm returns *false* indicating the program did not type check.

5.3.3 Termination of the algorithm

In this section I will make use of properties of partially ordered sets and some lattice theory. For further details and complete proofs please see [NNH99, Appendix A].

For this algorithm to be terminating, we need to ensure that the application of the annotators to the environment will eventually stabilise on a fixed element (such that $\Gamma = \Gamma'$ will be satisfied in the first loop), the annotator, annotation and type checker sets are all finite in size and that all annotator and type checker functions are terminating.

If a partial ordering ($\sqsubseteq$, a transitive, reflexive and anti-symmetric relation) is defined between environments for a given program (Γ^π, Γ'^π, etc.), and all the annotators in the framework are required to be monotonic with respect to the environments (i.e. $\forall \alpha \in \mathcal{A} : \alpha(\Gamma^\pi, \pi) = \Gamma'^\pi \rightarrow \Gamma^\pi \sqsubseteq \Gamma'^\pi$) then the application of different annotators to the environment in the first loop will form a chain of environments. E.g. (assuming α^π is an annotator that has been curried with its program already) for $\alpha_1^\pi, \alpha_2^\pi \ldots : \Gamma_1^\pi = \alpha_1^\pi(\lambda x.\emptyset), \Gamma_2^\pi = \alpha_2^\pi(\Gamma_1^\pi), \ldots$ such that $(\lambda x.\emptyset) \sqsubseteq \Gamma_1^\pi \sqsubseteq \Gamma_2^\pi, \ldots$.

To continue, we now need the following result, which we prove:

Lemma 5.3.1. *Within a framework instance, the number of possible environments will be finite.*

Proof. This is a consequence of the finite size of programs and the finite restriction placed upon Ψ.

- The set of `SExp` within the program will be of finite size, and so the domain of each environment will have the same size ($|\texttt{SExp}|$)
- The set of all possible annotations (Ψ) must be finite as a restriction
- $|\mathcal{P}(\Psi)|$ will therefore also be finite ($|\mathcal{P}(M)| = 2^{|M|}$)
- In an environment, each $\mathcal{S} \in \texttt{SExp}$ may map to any element in $\mathcal{P}(\Psi)$
- For a mapping of m keys to any of n values, the number of possible maps will be n^m
- There will therefore be $|\mathcal{P}(\Psi)|^{|\texttt{SExp}|}$ possible environments, which is a finite number

□

Since the number of possible environments is finite, if a partial ordering exists, then the environments will satisfy the Ascending / Descending Chain conditions ([NNH99, Appendix A, Lemma A.6]). This means all chains will eventually stabilise, and so the first loop of the algorithm will eventually stabilise upon some environment.

It now remains to actually define the partial ordering $\sqsubseteq$ and give restrictions that annotators must obey in order to be monotonic with respect to environments.

5.3.4 Partial ordering on Environments

For a given program π, the environments mapping the sub expressions (`SExp`) of that program to a set of possible annotations (Ψ) are partially ordered as follows:

$\sqsubseteq :: \mathcal{P}(\Gamma) \times \mathcal{P}(\Gamma) \rightarrow \{\texttt{true}, \texttt{false}\}$

$\Gamma \sqsubseteq \Gamma' \leftrightarrow \forall \mathcal{S} \in \texttt{SExp} : (\forall \psi \in \Gamma(\mathcal{S}) : (\exists \psi' \in \Gamma'(\mathcal{S}) s.t. \psi \leq \psi'))$

However, to use this partial ordering, equality of environments needs to be altered slightly. It becomes this:

$\Gamma = \Gamma' \leftrightarrow \forall \mathcal{S} \in \texttt{SExp} : (\forall \psi \in \Gamma(\mathcal{S}) : (\exists \psi' \in \Gamma'(\mathcal{S}) s.t. \psi \leq \psi')) \wedge (\forall \psi \in \Gamma'(\mathcal{S}) : (\exists \psi' \in \Gamma(\mathcal{S}) s.t. \psi \leq \psi'))$

I.e. that environments are now considered equal modulo subsumption of the annotations within them.

This gives a least environment ($\bot$) (which is the initialization environment in the algorithm) of $\lambda x.\emptyset$, and a maximal element ($\top$) of $\lambda x.\Psi$.

Lemma 5.3.2. *In the context of a framework instance, the relation $\sqsubseteq$ is a partial ordering over environments.*

Proof. To prove this, we must show that the relation $\sqsubseteq$ is Reflexive, Transitive and Anti-Symmetric.

- Reflexive: For any Γ show $\Gamma \sqsubseteq \Gamma$.
  For any $\Gamma \in \mathcal{P}(\Gamma)$ then $\forall \mathcal{S} \in \texttt{SExp}, \forall \psi \in \Gamma(\mathcal{S}) : \psi \in \Gamma(\mathcal{S}) \wedge \psi \leq \psi$.
  By this definition we have $\Gamma \sqsubseteq \Gamma$ as required.
- Transitive: For any $\Gamma, \Gamma', \Gamma'' \in \mathcal{P}(\Gamma) s.t. \Gamma \sqsubseteq \Gamma' \wedge \Gamma' \sqsubseteq \Gamma''$ then show $\Gamma \sqsubseteq \Gamma''$.
  By the definition of $\sqsubseteq$ we know that:
  $\forall \mathcal{S} \in \texttt{SExp}, \forall \psi \in \Gamma(\mathcal{S}) : \exists \psi' \in \Gamma'(\mathcal{S}) s.t. \psi \leq \psi' \wedge \exists \psi'' \in \Gamma''(\mathcal{S}) s.t. \psi' \leq \psi''$.
  Since $\leq$ is a partial ordering then we know that $\psi \leq \psi''$, i.e. $\exists \psi''' \in \Gamma''(\mathcal{S})(as \psi''' = \psi'') s.t. \psi \leq \psi'''$
  Hence $\Gamma \sqsubseteq \Gamma''$ as required.
- Anti-Symmetric: This follows directly from the definition of equality of environments.

□

5.3.5 Restriction on Annotators

With the partial ordering defined up to the subsumption relation on annotations, we now need to ensure that all annotators are monotonic with respect to environments. The following theorem is the main result of this section.

Theorem 5.3.3. *The environments that the annotators return must map each sub expression to the set of annotations in the input environment, or a superset thereof, modulo subsumption of annotations.*

Proof. A simple consequence of the construction of $\sqsubseteq$ and the definition of monotonic functions. □

Restrictions

To restate formally:
For a framework $\mathcal{F} = (\mathcal{A}, \Psi, \pi, \Omega)$

- $\mathcal{A}$ is finite in size.
- $\forall \alpha \in \mathcal{A} : \alpha$ is a terminating function.
- Ψ is finite in size.
- Ω is finite in size.
- $\forall \omega \in \Omega : \omega$ is a terminating function.
- $\forall \alpha \in \mathcal{A} : \alpha$ is monotonic with respect to $\sqsubseteq$.

5.4 Incremental Type Checking

With the algorithm as described, it will be necessary to re-type check the program using an empty starting environment every time the AST for the program changes. However, if the annotations are able to be linked to other annotations or parts of the AST, when the AST changes the annotations that depended upon those parts of the AST (and the annotations that depended upon those annotations etc.) can be removed, but the unaffected annotations may remain. In this way, as the AST for a program evolves, type checking doesn't need to start again from scratch, but can reuse existing information in parts of the AST that don't need to change.

The notion of a change in the AST also needs to be propagated up an AST from the leaves to the root. A simple way of doing this is to say that all ancestor nodes of any changed part of an AST are also marked as changed, and so all annotations on the path from the changed leaf to the root of the AST are removed. This could seem like an excessive number of annotations to be removed, and further work should look at ways of optimising this.

Also, annotations placed on the AST by a human user would need considering in light of these possible removals. It is quite likely that they should be persisted across AST changes, or options could be provided to allow them only to be deleted in certain circumstances.

5.5 Other Optimisations

Annotators are able to read the syntactic structure of an AST, and the annotations upon it. A given annotator, however, will likely only look for the presence of one or two types of annotation on the AST. If, in a given loop of the annotating part of the algorithm, no annotations of the types that the annotator reads are added or removed from the AST , then it is not necessary to re-execute that annotator (from the point of view of the annotator, the environment passed to it will not have changed, and as annotators are functions the returned environment should not have changed).

The algorithm could also be optimised for Non-local type analysis. Generally there is some form of abstraction or interface that means some areas of code cannot see the structure of other places. For example class and method nodes with annotations give an interface, and code in other classes

shouldn't see the code within the method nodes. Instead of having a single environment, there could be one environment per top-abstraction (e.g. class). The type checking framework can disallow other annotations having a dependency on any nodes that are children of the bottom dependency (e.g. method). This would give some form of notion of an interface and encapsulation of code.

5.6 Summary

This chapter gave a formalised version of the algorithm used to type check programs using a pluggable type system. This formalised algorithm has been implemented as part of this project, as is explained in the following chapter.

Chapter 6

FLEECE

6.1 Introduction

In order to demonstrate the ideas of this project in a concrete manner, a tool entitled FLEECE has been developed. It is a framework which allows for the construction of program abstract syntax trees based upon a user provided grammar, and then applying type annotators and checkers specific to that grammar to the AST in an incremental manner, using an enhanced version of the type checking algorithm given in chapter 5.

6.2 Informal Specification

- Take a definition of a program's defining grammar and present a way of editing instances of that grammar.
- Present a view of the in progress AST to a developer so that all parts of the AST can be identified.
- Allow type annotators and type checkers associated with the grammar to be used incrementally to type check the program as it is being developed.
- Allow a user to also add their own annotations to the programs being developed.
- To provide some form of feedback indicating which annotations have been placed on the certain "nodes" in the AST, and which (if any) nodes fail type checking.
- Explore incremental updating of the annotations as the program is developed, as opposed to re-annotating from scratch on all changes of the program.

6.3 The Editor

A structured way of editing the ASTs was wanted that would allow the program to remain syntactically correct and persistent across changes made, even if it was incomplete. This would facilitate some investigation into incremental type checking if time permitted.

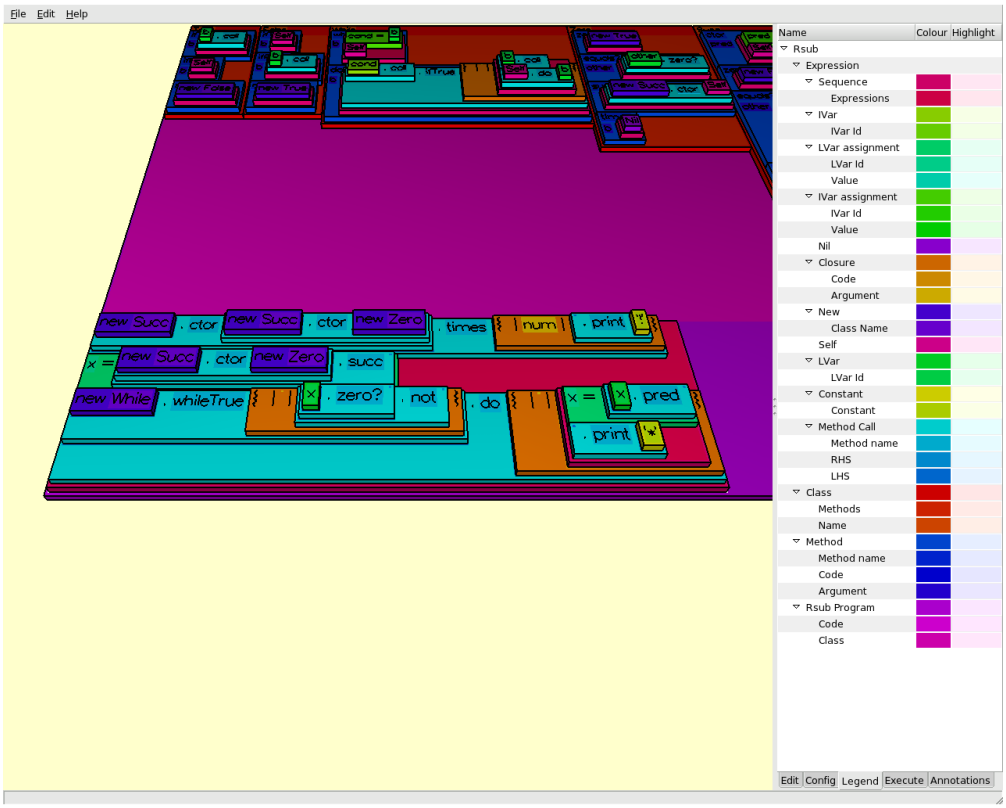


Figure 6.1: A screenshot of the FLEECE application. The abstract syntax tree of a program is visible in the main pane of the application, and the legend panel is visible, showing the different colours of the different nodes in the AST.

If a more traditional text based editing approach had been taken, then there would be the difficulty of maintaining the high level representation of the program across changes in the source text. Also, some textual changes would likely render the source unparsable and so no model would exist to aid the programmer at those points during the editing process.

Displaying the annotations on a pure textual input system could also be difficult as there may be implicitly created AST nodes in the textual input that are not represented in the text, and so there is no handle to be able to select them to see them.

6.3.1 Nodes and Children

The tool was written to be abstracted away from any particular language's grammar. Each instance of a grammar construct (method call, instance variable access, class, etc.) are represented in the tool by *nodes*. The grammar constructs describing these instances are modelled by *node factories*. Sub grammar constructs in non terminals of the grammar are modelled by *children* in the tool. Children can contain other nodes, representing the sub-expressions of the grammar they point to. Alternatively children can contain text if in the grammar they would contain class/method/identifier/other names that are not modelled by the structure of the grammar.

For example, if the grammar has a BNF production rule similar to `[methodcall]: expr[rec].methName expr[arg]`, then there will be in the tool a node factory for `methodcall` nodes. Each `methodcall` node will have three children, one named `rec` that accepts `expr` nodes, another similar child named `arg`, and a child that accepts text called `methName`.

In the grammars accepted by FLEECE, children can be restricted to accept certain numbers of children. There are four such constraints: *required* (exactly one node/text item must be present), *optional* (exactly one node/text item may be specified), *any* (any number of nodes or text items may be specified), or *one or more* (one or more nodes/text items must be present).

A programmer can add or remove the appropriate types of nodes or text to or from children (subject to the numerical constraints).

6.4 Implementation Details

The FLEECE editor has been implemented using the dynamic programming language RUBY. This was chosen because it is a highly expressive and flexible language, and has many powerful features as described in section 2.5. For the GUI, the Gtk2 toolkit was used, with the OpenGL extension (Gtk2GlExt) used for creating a widget that visualised the AST. These are mature and powerful graphical toolkits, and the RUBY interfaces to them proved stable and reliable throughout development. The actual construction of the GUI panes was using the Gtk2 Glade2 technologies. This entailed using a graphical editor to construct the look of the GUI, and then saving this out to an XML descriptor file. This file is parsed at runtime to reconstruct the GUI, and some management code was written to route widget events to the appropriate places in the code-base.

The application was written in a mostly test-driven style, the major exception being the testing of the GUI (which was done by hand). However the use of a visual editor for the GUI, and much of the user-interaction GUI code being kept small means it is a reliable application.

For the curious, the codebase is over 9200 lines of RUBY code (not including grammar descriptions and build-files). Around 3800 lines of code are in tests, and over 5400 source. These were all written / developed by the author.

Following is a description of the major components of the FLEECE application, which are outlined in Figure 6.2

6.4.1 Language Manager

FLEECE is not tied to being an editor for any one particular programming language. For any particular invocation of the application, the appropriate language definitions need to be loaded. These definitions consist of a required grammar description prescribing how programs are structured. Also associated to a language are optional features such as type checkers, type annotators, data providers, program executors, and a custom icon.

With the exception of the grammar file and icon, the language manager reflexively loads and maintains a list of classes for the optional features on a per-language basis. The classes are discovered by a simple filename-is-classname convention in prescribed directories. In order to simplify development, issues such as name collisions and security / trust of reflexively loaded classes has been ignored.

An example filesystem layout for the features of $\mathcal{R}_{sub}$ is given in Box 1. The directories holding annotators, executors, etc. are all walked automatically by FLEECE on startup, so no definition or configuration file needs to be updated as new annotators, typecheckers etc. are added; the application simply needs restarting.

Although this report primarily concerns itself with the $\mathcal{R}_{sub}$ language developed for demonstrating the ideas of this project, some other simple language descriptions where created for testing the application; for example the simple WHILE language of [NNH99], and the lambda calculus.

6.4.2 AST Model

The AST Model manages both the internal representation of a grammar file, and the program instances that are generated from that grammar by the user. After these internal representations have been loaded, the AST Model presents both view and edit interfaces to the AST of the current program being edited. In addition, the AST Model allows other classes to subscribe to it and be informed upon any change to the view of the model, or to specific node added/updated/removed events.

The grammar file loaded by the Language Manager is processed by an AST node factory builder, which builds AST node factories for each part of the grammar. The grammar for a language is written in a custom domain-specific-language which is really valid RUBY syntax. The grammar descriptions express usual BNF-like terminals and non-terminals, but also how the children of a node should be visually laid out during construction, the numerical constraints on them, where visual hints (such as the "." in method calls) should be placed and also friendly names for the nodes and their children that are displayed in the editor.

The grammar file that is used for $\mathcal{R}_{sub}$ is given in Appendix A.

The view interface is based around the notion of a selected node (and optional selected child) in the current program AST. It provides operations to navigate an AST that are tailored for a GUI to use. For example, move selection to the children of this node, select the next/previous child, "enter" a child (select the first of the nodes it contains), select this nodes sibling, etc. It also allows access to the factories that define nodes in order to get information such as the factory's user friendly name.

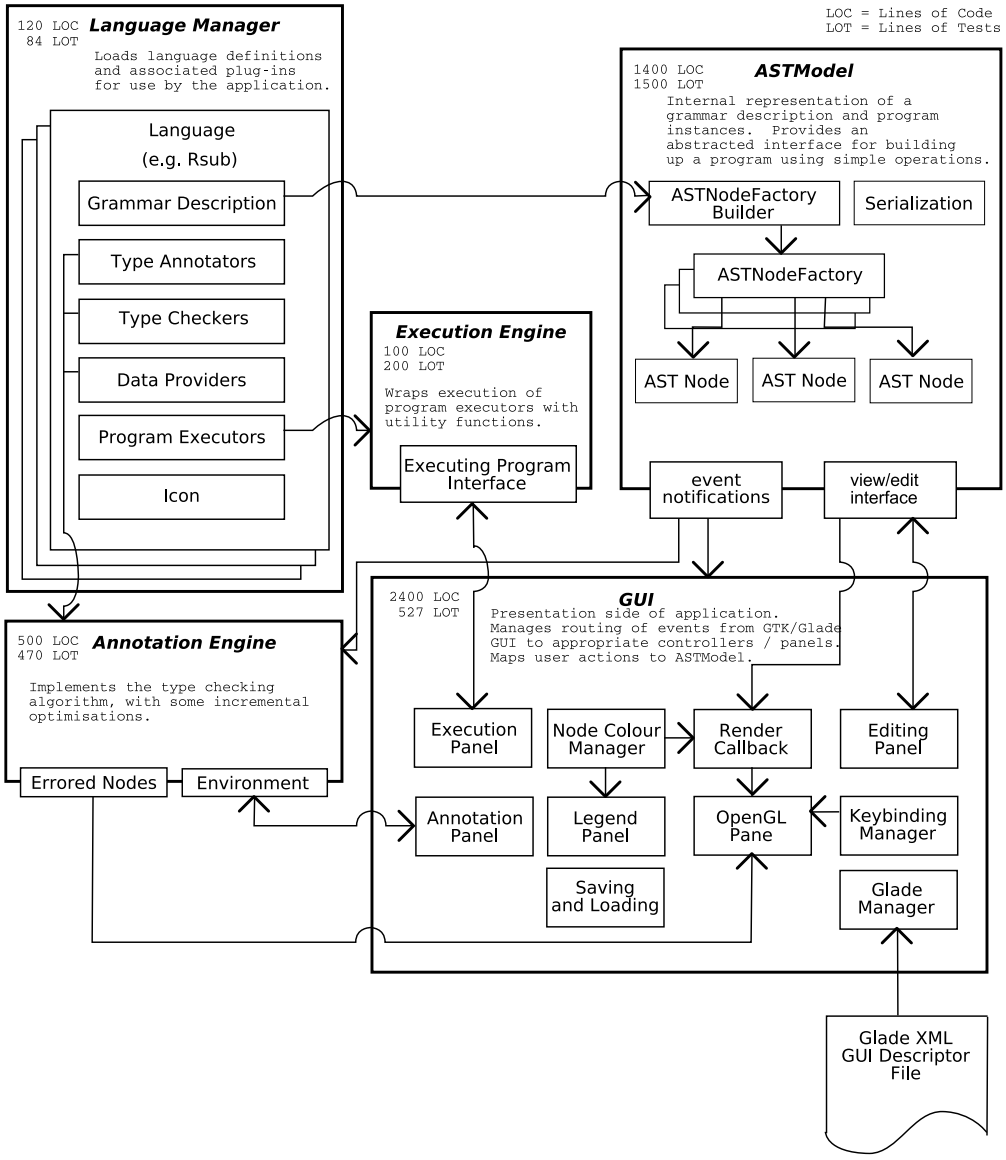


Figure 6.2: Overall structure of the components of FLEECE.

```
lib/
|-- ASTDefinitions/
|   |-- Rsub/
|   |   |-- Rsub.rast
|   |   |-- annotation.config
|   |   |-- annotators/
|   |   |   |-- LVarAnnotator.rb
|   |   |   `-- ...
|   |   |-- checkers/
|   |   |   |-- LVarChecker.rb
|   |   |   `-- ...
|   |   |-- dataproviders/
|   |   |   |-- RsubSymbolTable.rb
|   |   |   `-- ...
|   |   |-- executors/
|   |   |   |-- RsubExecutor.rb
|   |   |   `-- lib/
|   |   |       `-- rsub_exec_walker.rb
|   |   `-- icon.png
|   |-- WhileLanguage/
...
```

Box 1: The filesystem layout of the $\mathcal{R}_{sub}$language

There is close to a one-to-one mapping between the actions a user makes to navigate the AST in the GUI and the operations the view interface provides.

The edit interface works in the context of the currently selected node and child from the view interface. It presents visitor-pattern like callbacks for the currently selected child, depending on whether nodes or text may be added / removed (or both) from it. The code executed by these callbacks can then add/update/delete nodes/text in the currently selected child. It also can be queried to give the node factories which may be used to create nodes (if appropriate) on a particular child.

Mutating actions of the tree must always happen upon a child of a node that is selected. For a node to be deleted, the appropriate child of its parent node must be selected, and a delete command issued to the edit model (optionally specifying the index of the target node to remove if it is in a child that accepts arbitrary numbers of nodes). This means that there is always one root node that can never be removed from a program, as it has no parent node to select.

The AST Model supports two types of listener to changes in the AST tree or the model's view state. The first is a generic listener pattern which will update when any change to the AST Model takes place, such as a change of currently selected node or some edit made to the tree. This is used by several of the panes in the GUI and other places to keep the GUI information displays in sync with the underlying model. The second will fire one of three events on nodes being edited, updated or deleted. This is used by the Annotation Engine to keep annotations in sync with an evolving tree.

Two of the invariants of the AST Model is that there are no cycles in the AST, and that the same (i.e. pointer equal) node (and any associated sub-nodes) does not appear twice in the graph. The

first is precluded to make sure that the AST is always finite in depth, which means visitor-pattern walking of the tree is a safe and simple thing to do.

Allowing identical nodes in the tree would be an interesting extension, and was (briefly) considered. It was ruled out due to the added complexity it would cause in some parts of the implementation, particularly for the GUI (for example laying out nodes), and also tree-walking the nodes. Also, any advantages it may have for a programmer writing a program (reuse of some code in two places that should stay the same) would be more indicative of code that should be refactored.

6.4.3 Annotation Engine

The implementation of the type checking algorithm given in chapter 5 is done by the Annotation Engine. This subscribes to the node added/updated/deleted events from the ASTModel and uses the language-specific data providers, type annotators and checkers to type check the current program being developed. The type annotations placed by the annotators (or by a programmer) are kept in environments which are passed to the GUI for displaying, along with any nodes which the type checkers deem to be in error.

The optimisations discussed in section 5.4 are also implemented. The top-abstraction and bottom-abstraction nodes are given in a configuration file in the current language's definition folder. In the case of $\mathcal{R}_{sub}$ these are class and method nodes respectively. However time has not permitted an example type system to test or demonstrate the non-local optimisation. This is definitely work that should be pursued.

Data providers give the annotators and type checkers derived information on the AST of the program that can be shared as opposed to having them recompute it. For example symbol tables mapping class names to the node where that class is declared. They are transparent from the point of view of the type checking algorithm as they do not interact with annotations and are therefore an optimisation for efficiency. Since data providers should only change their internal state when the AST is structurally modified, they are only re-computed after AST changes and are re-used throughout any one run of the type checking algorithm.

Within the Annotation Engine, it is in environments that the mapping from nodes in the AST to annotations is kept. The environments are passed to the annotators who may add to or query the annotations on a given node, and also the type checkers who may only query.

The environments also facilitate two optimisations of the type-checking algorithm implementation as given in section 5.4. The first is that during each loop of the annotating process, the environments keep track of annotations that have been added during that loop. As explained, this means only the annotators which can understand these annotations will need to be run in the next loop.

The second optimisation is that there is not a single environment, but several, one for each top-abstraction node in the code (in the case of $\mathcal{R}_{sub}$ there is one environment for each class). This is to help protect encapsulation, and allow for future possibilities including loading/merging read-only environment definitions for library code. In order to simplify the presentation of the environments to the GUI there is also a wrapper class that acts like a "global environment", allowing a lookup of the annotations on any node in the tree without needing to discover which sub-environment to consult or needing to merge all the environments together into one big one. This wrapper class discovers the correct sub-environment for the requested node and delegates to it.

Annotations are implemented as mappings from keywords to instances of certain internal classes (mainly `String`). When annotators or type checkers query the annotations, they can specify a

pattern to check against. The checking of a pattern p against an annotation a is such that every key in p must have a corresponding key in a, and that the value attached to the key in p must either be equal to the value in a, or if that value is a class in p, it must be an instance of that class in a.

There is an implicit subsumption relation here, such that if two annotations are viewed as a set of $(key, value)$ tuples, annotation a subsumes annotation b if $a \supseteq b$. This is a subsumption relationship that transparently works to all annotators, given the mechanism for annotation querying given above. This means that if a property holds with annotation a, and b subsumes a, then that property must hold with b.

The implicit subsumption relation, however, does not allow for subsumption based on just the values in the $(key, value)$ tuples varying. Although allowing this would greatly add power to the system, a way of declaring the partial ordering between annotations (in a system where annotators can be plugged in and possibly out), and taking it into account when querying them would need developing. Sadly time did not permit this, and it should be considered for future work.

When annotators are instantiated by the annotation engine, they tell the annotation engine which annotations they understand by specifying the appropriate annotation patterns and passing them to the annotation engine. This information is then used to optimise the annotating loop of the type checking algorithm, as described.

The GUI also has limited access to put annotations on the tree, marked as coming from a human operator. Adding these annotations will also initiate of a round of type checking for those annotators that would understand the annotation.

After the annotations have been placed on the tree, the type checkers are run. Type checkers are able to tell the annotation engine that nodes are in error. These nodes are kept and passed to the GUI on request.

6.4.4 Execution

Programs can have associated with them executors, which take an AST, and execute it. These executors are run in a separate thread by the execution engine, and may interact with the parent application via in, out and error streams. The GUI provides a panel through which a developer can see the output of a process and possible give input.

6.4.5 GUI

The GUI component consists primarily of handlers for the different panels (edit, annotation, execution etc.) in the application, and also the pane that visualises the AST.

In order to visualise the AST a box-in box design was used. A node would contain all its children. A nice feature of this is that if the border of the nodes are collapsed to 0 then the AST representation reads like the original source code.

Alternatively an expanding parse tree representation could have been used, but this can be more complicated to read as

Visual features were leveraged to make the presentation of the AST potentially more useful to a developer:

- *3D:* This was used to make the program look distinctive and provocative. It also accents the different parts of the AST, and makes it clear where there are different nodes and what are place-holders for children. It was imagined that a further benefit of using 3D would be to improve (and provide other) ways of finding particular pieces of code in a 3D landscape. However programs large enough to utilise this theory have not been developed with this software. There is also literature expressing that the benefit of recall in 3D environments is no better or worse than using a pure 2D environment[Coc04]; and that for remembering general items, using names is generally better than a spatial location[JD86] (2D or 3D).

- *Colour:* This was used to differentiate the different types of node and children. Each node and their children are allocated a colour depending on their type. This can (after the colours have been learned) make it easier to see at a glance what the structure of a program is. The colours are allocated by working out how many distinctive node/child types there are, and then dividing a HSV colour cone space into that many parts[Wik06].

- *Visual Hints:* There are many conventions to modern programming languages; for example method invocation is usually specified with a "." following the receiver, or assignments feature an "=" in their constructions. While these are usually necessary to enable a grammar that can be parsed, they are technically unnecessary in the system developed (nodes can be told apart purely by colour for instance, or by querying them for their name). However, in order to keep the learning curve of the application down, visual hints such of these can be specified to be placed in the nodes in the annotated grammars that FLEECE reads.

- *Layout:* The layout of the children in a node are positioned, and how children that can accept arbitrary numbers of nodes position them, can be specified in the grammar files that FLEECE accepts. This means that other traditional programming conventions, such as statements being placed one above the other, or assignments being read left to right, can also be presented in their traditional format. Again, as with visual hints, this isn't necessary, but for familiarity to the target audience it was taken into consideration.

6.5 Summary

This chapter outlined the application, FLEECE developed as part of this project, and explained some of the design and implementation decisions made as part of that implementation. This was the resulting product from this project and includes an implementation of the type checking algorithm, type systems and example language that have been developed during this project. The following chapters evaluate and conclude this report.

Chapter 7

Evaluation

This project has been an exploration of a new idea, and as such there is no direct counterpart or predecessor to it that it can be evaluated against. However, many of the ideas within the project have been variations or extensions of other ideas, so facets of the project can be compared.

7.1 Type Checking

Traditional type checking is not always cast as a two stage algorithm of annotate and then check, but instead a case of if the program can be annotated then it passes the type check. However the idea of annotating sub-expressions in a program with properties that hold, and then checking if things are in error with those properties is closer to a generate-constraints and then check them style of type checking. This idea is used by Palsberg and Schwartzback in [PS91] for discovering which classes the receiver of methods may be in a simple dynamically checked object oriented programming language. Their algorithm is similar in style to the one developed in this project, the program is annotated with some constraints and then these constraints are iterated to a fix point where they are then checked. As a comparable piece of work to some aspects of this project, it is highly likely that their specific type system could be cast into a form compatible with the pluggable type checking algorithm.

One of the underlying assumptions of the pluggable type checking algorithm is that annotations may features on any part of the abstract syntax tree of a program. Since most of these annotations are likely to be inferred by annotators this is not a problem for using it for existing dynamically typed languages. However, some annotators may need a human to place an initial annotation in order to start generating constraints (perhaps a `final` annotation to denote a variable should be final), in these cases there is no restriction from the point of view of the algorithm as to where these annotations could be placed. This means that a language that only supports manually place annotations at some points in the source code can work with the algorithm, but lose some of the flexibility of human annotations anywhere.

Recently, annotations have been added to some of the statically typed languages; JAVA and C♯ (where they are called attributes)[1]. Many of the editors for these languages have plug-in projects using the annotation systems to provide sophisticated source level analysis. INTELLIJ IDEA, as mentioned in the background to this project, can check and analyse methods using annotations

[1]For the more sophisticated dynamically typed programming languages, adding source level attributes is also a straightforward meta-programming task.

marking them as never returning `null`. For VISUAL STUDIO / C♯ there is an extension called XC♯ [2] that uses annotations to guide where custom written analysis code should be run, and to provide hints to it. This analysis code can then interact and display information back in the VISUAL STUDIO editor. These analysis codes can do more than just error checking however, they are able to alter the generated bytecode for classes (for example annotations to mark obfuscation or add extra run-time checks). In comparison this project focused only on the use of annotations for type checking. It did not consider further uses for those type checks. However there does not appear to be comparable frameworks for annotations for use with purely dynamically typed programming languages, which was the focus of this project.

Some comparison can be drawn between some of the aims of the pluggable type systems presented and traditional *lint* or warning modes of existing languages. These would look for bad patterns of usage (e.g. method invocations with incorrect arities) and give errors if they were discovered. They generally operate to raise an error if something bad will happen, but not guarantee that nothing bad can happen. For the users of dynamically typed programming languages, this is usually the ideal. The framework for type systems given should (in theory) allow for lint-like type checking to happen, but also optionally the more restricted type also.

7.2 $\mathcal{R}_{sub}$

The formalised language given is a particularly simple variant of a dynamically typed language, and while it does model a subset of RUBY with features such as no pre-declaration of instance or local variables and closures, it is lacking in many other features such as inheritance, reflection, run-time code evaluation and libraries. The language was kept simple to enable a demonstration and motivation for pluggable type systems to be developed within the time-frame available. The choice of formalising a subset of RUBY, as opposed to reusing an existing one (such as a pure object calculi [AC96] or extensible JAVASCRIPT subset [AGD05]) meant that an interesting language feature to the author (closures) could be explored. The example type system developed using the pluggable type checking framework makes use of closures in the process.

The formalisation of the language would also allow some analysis of the sample type systems developed, particularly proofs of the properties they represent, unfortunately time did not allow for this to take place. This project would be greatly improved if the sample type systems had been formalised, and some proof of guarantees they give when applied to a program shown.

$\mathcal{R}_{sub}$ programs are executed by converting them to an equivalent RUBY version. Due to limited time, the translation does not take into account possible name clashes with existing methods, objects or classes in the RUBY standard library. (An observant reader will note that this was taken advantage of to capture output from the various $\mathcal{R}_{sub}$ examples spread throughout this report). Since these name clashes can be solved with some equivalent of α-conversion of $\mathcal{R}_{sub}$ class or method names, it is not a major issue, but it could be improved.

7.3 FLEECE

The tangible product that has resulted from this project, FLEECE is a working, and stable application. It was developed to demonstrate the idea of annotations being applicable to any node in an

[2] http://www.resolvecorp.com/products.aspx

AST, and that those annotations are visible to a programmer (in some way). This it does do fairly successfully. However, it is a prototype application, and could be improved.

As it was developed around editing and building up generic abstract syntax trees, it has a user interface that is not tailored to general programming. For example, expressing some code that would textually look like `x = A().b(c.d())`, requires a programmer to know that they want to create an assignment, and on the right hand side of the assignment a method call, on the left hand side of the method call, a new object, and on the right another method call on a local variable; essentially constructing the inverse of the parse tree for the expression. As will be discussed in the future work section (section 8.2) there are possible ways to improve this.

What is interesting to note, is that while writing programs in FLEECE, it is very quick and simple to create the high level abstractions (classes, methods); and that because the AST is syntactically robust, classes and methods can be prototyped without needing to fill in all details such as names or arguments. However it is the details of the code, such as expressions, that are irritating to write, due to the reasons given above.

Improvements could also be made to the way a user navigates around the AST. Currently there is no way to just click on a node and take selection there, and instead a slightly confusing keyboard model for navigation is used. This has the disadvantage that to select a node that is potentially far away, the common parent node has to be navigated up to, and then selection changed to move down the branch to the target node. This movement by indirection is time consuming and irritating if a programmer wants to jump quickly between pieces of the code.

The syntactic robustness of the AST that allows it to be type checked and annotated even though it is incomplete is a great strength. Many current editors (e.g. the JAVA editor ECLIPSE) are not able to semantically analyse all statements within a file if there is a syntax error. While the situation is improving with these editors, having an AST structure means code can be prototyped and type checked, even to the level of a developer being able to think "I know I need to make a method call on this object with this argument, but I don't know the name of the method yet, I'll leave it blank", and still getting type analysis of both the receiver and the argument, despite the name just being there is a great benefit over a more traditional editor requiring either a dummy name to be inserted (which will be highlighted negatively as an error when it is really a task to be done), or the developer needing to switch train of thought to create and find out the method name as opposed to dealing with the problem they were working through.

Although FLEECE is robust like this, and does indicate when a particular node is selected if it is *complete* (i.e. is a fully valid instance of the grammar with nothing missing), it currently does not have a way of visually displaying this information (although it would be trivial to add).

The interpretation of the visual display of the programs within FLEECE can only be a subjective one. However in general when people are first exposed to images of the application, the nature of what they are seeing does need to be explained that this is an abstract-syntax-tree view of a program. After this explanation, people (programmers) generally do understand what is going on. As with editing the application, viewing a graphical AST does help with quickly grasping the high level abstractions, but can add noise for the details.

After using the editor for some time, the colours of the nodes also do help with recognition for what they contain (with $\mathcal{R}_{sub}$ the yellow colour of the closure nodes is particularly distinctive), again however this is an entirely subjective analysis. The layout of nodes combined with their colour is particularly useful for quickly discriminating a method definition from a class (for example).

It should be stressed that the primary aim of developing the FLEECE application was to demonstrate and motivate visually the ideas of incremental pluggable type checking, and that the visual programming side of the application was a by-product.

Aside from the visuals, the prototypical nature of FLEECE coupled with its implementation being in a relatively slow, interpreted language means that the application can start to take a noticeable time to respond as the size of the program being edited grows. It is also expected that as the number of type annotators, annotations and checkers grows that are plugged into the system that the response times would also drop. This is due to the type checking happening in the user thread in-between actions and program redraws and could be improved.

Chapter 8

Conclusions and Future Work

8.1 Contribution

This project has made some contributions to the area of pluggable type systems, these were outlined in the introduction to this report (section 1.2) and are recapped here;

In order to present a working example of the theory, a dynamically typed programming language has been formalised (chapter 3). This language ($\mathcal{R}_{sub}$), is a restricted subset of the RUBY programming language. To make this language distinctive, it models the notion of a closure, but simplifies many other aspects; for example ignoring reflection, control flow and inheritance.

Using $\mathcal{R}_{sub}$, an example pluggable type system was presented (chapter 4). The different parts of the type system (the annotators and type checkers) are introduced and their function in the example explained. The example demonstrates that type systems can be built up and improved in a compositional manner. The example also lead into a discussion into the role of a programmer placing annotations; and the different use cases for type inference compared to type checking.

Building upon the example pluggable type system given, a formal description of the components of the type systems was described, before an algorithm for type checking in such a system (chapter 5). The properties required to ensure this type checking algorithm would terminate were given.

Some discussion and suggestions outlined how the type checking algorithm could be improved for use in an incremental setting. This would allow type information to be reused while a program was being edited, as opposed to requiring all the type information be thrown away.

Finally an application was developed that allows programs to be written in it by a programmer (chapter 6). The type checking algorithm was implemented in the application, along with some of the incremental type checking extensions that have been discussed. The annotations placed on programs (by annotators or a developer) are able to be seen by a developer. To demonstrate this, the language $\mathcal{R}_{sub}$ is available in the application, and some of the type systems described have been implemented to work with it.

8.2 Future Work

As an early exploration into the idea of pluggable type systems for dynamic programming languages, this project has tried not to answer every question, but instead discover which questions are interesting to ask.

8.2.1 Types and Code

- *The nature of type checking:* There are several ways of using the typing framework, as touched on in section 4.4. Annotators can create generate constraints and these can be checked, or the programmer can give an annotation asserting a property should be true and whether it is or not could be. Also with constraint based type checking, what is the cause of the error could be one of several different places depending on how the constraints are formed. This could also be related to the different types of programming code there are (library, wrapper, application etc.) Some research into the different use cases for each type of type checking, and benefits / disadvantages of each (and possibly others) could be interesting.
- *What is an error?* If an error is detected in dead code, this will not be an error that stops a program from running. Some work could be done into reviewing
- *Non-local type analysis:* There is scope for improving how non-local type analysis could work. However it is likely that it is a problem that needs investigating in the context of a specific language and then generalising, as the notion of what is an appropriate "local" varies by language.
- *Incremental type analysis:* Further optimisations could be made for the process of incremental type analysis using the pluggable type checking algorithm. There is also much scope for exploring alternative ways that this could work.
- *Other type analysis:* There are many exotic type systems currently being researched; investigating how easy they are to cast in the type framework given, and what the potential benefits are.
- *More powerful language:* The $\mathcal{R}_{sub}$ language was heavily restricted. One of the aims of this project was to allow developers using "real world" dynamically typed languages to have the benefits of some static type checking. Exploring possible pluggable type systems for a current mainstream language would be a good next step.
- *Runtime reflection of static annotations:* Being able to reflect at runtime on the annotations placed on parts of the code tree could be a useful ability (certainly its currently available in JAVAand C♯), however it does pose the problem that the runtime semantics of the application are then affected by which plug-in type annotators are run against the program.
- *Hybrid type checking:* In a paper by Flanagan ([Fla06]), he investigated Hybrid Type Checking, where type checks that could not be proved statically where turned into assertions checked at runtime. Could such an idea integrate into the world of pluggable type systems?

8.2.2 Program Integration

- *Plug-in type systems:* The project focused on plug in able type systems with the meaning of plug-in being an ability to turn them off and on. There are issues that should be addressed with the actual way a real-world system would allow type annotators and checkers to be plugged in. Problems to consider would be a way of declaring annotations and any subtyping / subsumption relations between them, namespacing annotations, and possibly even version comparisons of them. Also other issues such as security (would running a plugged in annotator require trusted code to be used, or could annotators / type checkers be represented in some safe language?) could be considered.

- *Annotation Guided Assistance:* Annotations provided by annotators or the user could guide or interact with a program editor in other ways. For example code completions on available variables or ways to quick jump around a code base to other annotation-indicated relevant places.

8.2.3 Usable Visual Application

- *New application:* The application, FLEECE, produced as a result of this project is intended as a prototype to explore the possibility of using pluggable type systems. A faster and more user-friendly application needs developing that can be used in the real world. This would need to consider the human-computer-interaction of a developer more closely.
- *Animation / Alternative Visualisations:* Many people have commented that the 3D visual display of program code could be used in other ways, for example an animated effect showing code reducing in a functional language, animations showing where the type annotators and checkers currently are when they are annotating the tree, or possibly even the usage of the code as a 3D environment people can walk around in (maybe to learn the code-base - a real guided tour of the code).

8.2.4 Other

- *Using the formalisation of $\mathcal{R}_{sub}$:* Currently the formalisation of $\mathcal{R}_{sub}$ is not used for proving the correctness of the type annotators / checkers based upon it. It is possible that a different style of proof or way of reasoning about the annotators / checkers may be necessary in this pluggable environment, and using the formalisation of $\mathcal{R}_{sub}$ as a starting point before abstracting this theory could be looked into.
- *Fixing the translation for $\mathcal{R}_{sub}$:* As discussed in section 7.2, the translation from $\mathcal{R}_{sub}$ to RUBY does not guarantee identical behaviour as to the operational semantics, owing to possible name clashes with the RUBY standard library, and closure-local-variable non-shadowing. Using RUBY's powerful meta-programming constructs it should be possible to correct this, or alternatively build a operational-semantically correct virtual machine for the language.
- *Other types of Language:* The focus of this project has been in the context of imperative, object-oriented dynamically typed programming languages, however other types of dynamically typed language could benefit from a similar idea; for example logic programs (PROLOG), or annotations as tactics for proof languages.
- *Swarm Theory:* Some analogy can be made between the annotators walking the AST of a program and depositing annotations based on the local environment (current syntactic structure and other annotations there) and other natural systems (e.g. ant colonies communicating using pheromone trails). Perhaps there is something there?

8.3 Conclusion

This project has demonstrated that there is potential for benefit from further research into the theory of pluggable type systems. It has motivated their use and shown how using simple type

annotators and type checkers, relatively complicated errors can be detected statically in an otherwise runtime-only typed language.

Also as a side effect this project does raise the further question of visual programming, or alternative ways of viewing code as it is being edited. With annotated information available at every program point, and a developer able to place annotations there themselves too, alternative ways of editing programs need to be investigated.

Appendix A

$\mathcal{R}_{sub}$ FLEECE Grammar

The following is the definition of the $\mathcal{R}_{sub}$ grammar which is used as input to FLEECE. It is valid RUBY code, and a good demonstration of a simple domain-specific-language within that language. The features of the grammar, such as allowing layout specification of the children of nodes, and where visual hints (in the form of labels) should be clearly visible.

```
declare :program do
  root
  display "Rsub Program"
  children( vertical(
       row(node(:class,:class,"Class")),
       req(node(:code,:expression,"Code"))
      )
   )
end

declare :class do
  display "Class"
  children( vertical(
       req(text(:name,"Name")),
       column(node(:methods,:method,"Methods"))
      )
   )
end

declare :method do
  display "Method"
  children( vertical(
       req(text(:name,"Method name")),
       horizontal(
    optional(text(:arg,"Argument")),
    req(node(:code,:expression,"Code"))
          )
      )
   )
```

```
30 end
31
32 declare :expression do
33   abstract
34   display "Expression"
35 end
36
37 declare :lvar do
38   isA :expression
39   display "LVar"
40   children(horizontal(req(text(:name,"LVar Id"))))
41 end
42
43 declare :methcall do
44   isA :expression
45   display "Method Call"
46   children(horizontal(
47         req(node(:lhs,:expression,"LHS")),
48         label("."),
49         req(text(:name,"Method name")),
50         req(node(:arg,:expression,"RHS"))
51        )
52    )
53 end
54
55 declare :ivar do
56   isA :expression
57   display "IVar"
58   children(horizontal(req(text(:name,"IVar Id"))))
59 end
60
61 declare :new do
62   isA :expression
63   display "New"
64   children(horizontal(label("new"),req(text(:classname,"Class Name"))))
65 end
66
67 declare :lvarAssig do
68   isA :expression
69   display "LVar assignment"
70   children(horizontal(req(text(:name,"LVar Id")),label("="),
71         req(node(:rhs,:expression,"Value"))))
72 end
73
74 declare :closure do
75   isA :expression
76   display "Closure"
77   children(horizontal(
```

```
78          label("{ |"),
79          optional(text(:arg,"Argument")),
80          label("|"),
81          req(node(:code,:expression,"Code")),
82          label("}")
83        )
84    )
85 end
86
87 declare :ivarAssig do
88   isA :expression
89   display "IVar assignment"
90   children(horizontal(req(text(:name,"IVar Id")),
91          label("="),
92          req(node(:rhs,:expression,"Value"))))
93 end
94
95 declare :nil do
96   isA :expression
97   literal
98   display "Nil"
99 end
100
101 declare :constant do
102   isA :expression
103   display "Constant"
104   children(horizontal(req(text(:name,"Constant"))))
105 end
106
107
108 declare :self do
109   isA :expression
110   literal
111   display "Self"
112 end
113
114 declare :sequence do
115   isA :expression
116   display "Sequence"
117
118   children(horizontal(
119          column(node(:code,:expression,"Expressions"))
120        )
121    )
122 end
```

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag, New York, 1996. 7.2

[AGD05] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *ECOOP*, pages 428–452, 2005. 2.5.4, 7.2

[Ayc] John Aycock. Aggressive type inference. http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html. 2.3

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993. 2.2.3, 2.2.4

[BI82] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proceedings POPL '82*, Albuquerque, NM, 1982. 2.3

[Bon] Daniel Bonniot. The nice programming language. http://nice.sourceforge.net. 2.3

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages. 1.1, 2.2.1, 2.2.2, 2.2.3

[Coc04] Andy Cockburn. Revisiting 2d vs 3d implications on spatial memory. In *CRPIT '04: Proceedings of the fifth conference on Australasian user interface*, pages 25–31, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. 6.4.5

[Edw05] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *OOPSLA*, pages 505–518, 2005. 2.3

[Fla06] Cormac Flanagan. Hybrid type checking. In *Proceedings POPL '06*, 2006. 2.2.2, 2.3, 8.2.1

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. 3.5

[GS] Andi Gutmans and Zeev Surask. Php programming language. http://www.php.net. 2.4

[Hei95] N. Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, pages 189–206, 1995. 2.3

[JD86] William P. Jones and Susan T. Dumais. The spatial metaphor for user interfaces: experimental tests of reference by location versus name. *ACM Trans. Inf. Syst.*, 4(1):42–63, 1986. 6.4.5

[Jet] JetBrains. Intellij idea. http://www.jetbrains.com/idea. 2.3

[LC05] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. 2.3

[Mata] Yukihiro Matsumoto. How ruby sucks. http://www.rubyist.net/ matz/slides/rc2003/. 2.5

[Matb] Yukihiro Matsumoto. Ruby programming language. http://www.ruby-lang.org. 2.4

[MD] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. http://pico.vub.ac.be/ wdmeuter/RDL04/papers/Meijer.pdf. 2.2.2

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992. 2.3

[NBD+05] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In *Software Composition*, pages 1–13, 2005. 2.2.3

[Net] Netscape. Javascript programming language. http://www.mozilla.org/js/. 2.4, 6

[NNH99] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. 5.3.3, 5.3.3, 6.4.1

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002. 5, 2.6.1

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991. 7.1

[RBFDD98] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference to support object-oriented reenginerring in smalltalk. In *ECOOP Workshops*, pages 76–77, 1998. 2.3

[Sal04] 1978-Michael Salib. Starkiller : a static type inferencer and compiler for python. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2004. 1

[Str] The strongtalk type system for smalltalk. http://bracha.org/nwst.html. 1

[Suna] Sun. Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. 2.3

[Sunb] Sun. Java bug parade. http://bugs.sun.com/bugdatabase/. 1

[Suz81] Norihisa Suzuki. Inferring types in smalltalk. In *POPL*, pages 187–199, 1981. 2.3

[TH01] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. pub-AW, pub-AW:adr, 2001. 2.5, 8

[vR] Guido van Rossum. Python programming language. http://www.python.org. 2.4

[Wik06] Wikipedia. Hsv color space — wikipedia, the free encyclopedia, 2006. [Online; accessed 8-June-2006]. 6.4.5

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999. 2.3