FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation and Implementation of an optional, pluggable Type System for Forth

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Gregor Riegler BSc.
Matrikelnummer 0703762

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 18. August 2015

Gregor Riegler	Franz Puntigam

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation and Implementation of an optional, pluggable Type System for Forth

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Gregor Riegler BSc.
Registration Number 0703762

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Vienna, 18th August, 2015

Gregor Riegler

Franz Puntigam

Erklärung zur Verfassung der Arbeit

Gregor Riegler BSc.
Ameisgasse 15/9, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. August 2015

Gregor Riegler

Kurzfassung

Normalerweise hat eine Programmiersprache genau ein Typsystem, das den Ausdrücken eines Programms Typen zuweist und deren korrekte Verwendung überprüft. Dabei beeinflusst das Typsystem die Semantik der Sprache, sodass die Typüberprüfung als Phase der Programmübersetzung nicht wegzudenken ist. Optionale Typsierung stellt eine Alternative zu dieser Kopplung von Programmiersprache und Typsystem dar und erlaubt damit optionale, verschiedenartige semantische Analysen eines Programms. Diese Arbeit konzentriert sich auf die Analyse statischer Typkonsistenz. Es wird dazu in Haskell ein optionales Typsystem für die stack-orientierte Programmiersprache Forth implementiert. Zusätzlich zu einer Literaturrecherche zu verschiedenen Ansätzen, sowohl statische als auch dynamische Typüberprüfung zu nutzen, dient die Implementierung speziell dazu, die Vorteile wie auch unvermeidbare Kompromisse und Limitierungen eines derartigen Typsystems aufzuzeigen. Im Besonderen wird die Frage behandelt, inwiefern die optionale Typisierung bei der inkrementellen Entwicklung eines Forth-Programms zu gesteigerter statischer Typkonsistenz führt, welche Sprachfeatures dabei unterstützt werden bzw. ein welches Maß an Typannotationen trotz Typinferenz notwendig ist. Dazu wird zuerst bestehende Literatur zur statischen Analyse von Forth-Programmen herangezogen, um eine theoretische Grundlage für die Implementierung der statischen Typüberprüfung zu gewinnen. Danach werden nach und nach Sprachfeatures wie Subtyping, Kontrollstrukturen, Referenztypen, Assertions und Casts eingebaut. In dieser Arbeit wird weiters als Neuheit sowohl die Integration von statisch typkonsistenten indeterministischen Stack-Effekten, als auch die Einbindung von objektorientierter Programmierung, Higher-Order-Programmierung und Compile-Time-Programmierung in den bestehenden Algorithmus zur Typüberprüfung demonstriert.

Zur praktischen Evaluation wird ein funktional korrektes Forth-Programm unter Konfigurationen steigender statischen Typkonsistenz überprüft und dabei werden Verstöße statischer Typkonsistenz ausgewiesen. Dabei zeigt sich, dass das optionale Typsystem bei der inkrementellen Weiterentwicklung des Programms zu gesteigerter statischen Typkonsistenz hilfreich ist und aufgrund der implementierten Stack-Effekt-Inferenz dazu nur wenig Hilfe in Form von Typannotationen benötigt. Typannotationen für im Programm definierte Wörter können oft weglassen werden, sind aber immer bei Gebrauch von Higher-Order-Programmierung notwendig.

Abstract

Typically, programming languages provide one type system which associates types to a program's expressions and checks their consistent use either at compile-time or at run-time. In any case, the type system influences the semantics of the language such that type checking is a mandatory process. In contrast, optional, pluggable typing introduces the possibility of running different checkers for diverse semantic analyses and can be considered as an alternative to today's prevalent mandatory typing and the tight coupling of a programming language and its type system. This thesis aims at exploring the design of a Haskell prototype implementation of optional, pluggable typing that provides checkers enforcing increasing static type safety for the stack-based language Forth. Alongside a literature research on various approaches bridging between static and dynamic typing, the prototype design and implementation serve to shed light on the benefits as well as the inevitable trade-offs and practical limitations of such a type system.
In particular, this thesis addresses the question of how optional checkers, supporting varying degrees of static type consistency, can be designed and implemented for practical use in Forth considering the trade-offs of a potential loss of expressiveness and the need of necessary type annotations. To this end, a stack effect calculus is derived from a literature review on static analysis of Forth programs as a theoretical basis for type checking and subsequently, language features as subtyping, control structures, reference types, assertions and casts are added to the implementation. It is the added value of this thesis to incorporate the above features together with a statically type consistent treatment of multiple stack effects, compile-time programming, object-oriented programming and higher-order programming into the same core type checking algorithm.

The resulting prototype can deal with the vast majority of the words of Forth's Core wordset and features a type checking algorithm of configurable static rigorousness, ranging from bare stack underflow and overflow checking to full-fledged static type consistency. The checkers are practically evaluated using the example of a small, functionally correct Forth program. The evaluation shows that those checkers of varying static type safety aid the gradual evolution of the input program towards a static stack discipline. This benefit comes at the expense of only minimal type annotations. Type annotations of words defined in the program can often be omitted but they are always needed in supporting higher-order programming.

Contents

CHAPTER 1

Introduction

1.1 Motivation

One of the most distinctive features of a programming language is its type system which associates types to a program's expressions. A type system gives rise to type checking, i.e. checking the consistent use of operations on values of given types. Consequently, many languages use type checking to detect faulty programs either at compile-time or at runtime, hence the categorization of statically and dynamically typed languages. Both kinds have their pros and cons as static typing provides static type consistency but impairs the language's expressiveness at the same time. However, soft typing, gradual typing and optional typing count among several approaches of mixing static and dynamic type checking in order to combine the advantages of both worlds.

Optional, pluggable type systems have been proposed by [Bra04] as a default in language design. Optional, pluggable typing introduces the possibility of running different checkers for diverse semantic analyses. It dissolves the coupling of a language and its type system and can be considered to be an alternative to today's prevalent mandatory typing.

Accordingly, using optional checkers for the problem of type consistency, thus enforcing varying degrees of static type consistency, looks like a trade-off between a program's expressiveness and type consistency which is worth investigating in this thesis.

1.2 Problem Definition

Since the challenges of type checking naturally depend on the features of the given language, optional typing needs to be analyzed in the context of a specific language. Optional typing being feasible only if a language meets characteristics as given in [Bra04], this thesis focuses on optional, pluggable typing for Forth.

There are several good reasons to choose Forth for an elaboration on the possibilities of such a type system.
Being a stack-based language, Forth features a pretty simple computational model, the *stack*, which is susceptible to static analysis ([Pö90, Poi91, SK93, Kna93, Pö02, Pö03, Pö06, Pö08]). Concepts such as *subtyping* and *object-oriented* programming are not at the core of Forth programming, thus the optionality of a type system can be analyzed in absence of those concepts (which are vital in many other languages) as well as including them. In particular, the retention of the untyped runtime semantics as required by [Bra04] is satisfied trivially as the untyped context of standard Forth goes well with that constraint.
In general, Forth does not have any type checks at all. Forth has a rich history of being used in performance critical, embedded programming contexts where the limitation of expressiveness forced by type checking is said to interfere with low-level programming. Hence, an *optional* type system makes sense for Forth: it introduces the possible benefits of static type checking whenever the programmer does not need that low-level expressiveness while it does not force it to be mandatory per se.

Analyzing optional, pluggable typing in the context of Forth, this thesis answers the following research question:

- How can optional, pluggable typing, supporting checkers of varying static type consistency, be implemented for Forth and what benefits and language restrictions or necessary type annotations does it involve practically?

1.3 Methodological Approach

1. *Comparison of static and dynamic typing.* Primarily, this step serves to clarify the pros and cons of those typing schemes. Secondly, it addresses the pragramatic trade-offs between type inference and type annotations given language features such as subtyping. In addition, the motivation for a combined typing approach is given.

2. *Literature study on combining static and dynamic typing.* The idiosyncrasies of soft typing, gradual typing and optional typing are analyzed. Implementation ideas for the prototype are gathered in that process.

3. *Literature review of the static analysis of Forth programs.* A stack effect calculus is derived from related works in order to find a theoretical basis for type checking and type inference for Forth programs.

4. *Haskell prototype design and implementation.* The stack effect calculus gets implemented and it is demonstrated how to integrate Forth language constructs providing static type safety.

5. *Varying the degree of static type safety.* The ability to vary the degree of enforced static type consistency is built into the prototype. As a result, different checkers can be created based on an according configuration.

6. *Evaluation of the use of the checkers.* Finally, sample checkers of increasing static type consistency are created for practical evaluation. An analysis of their application in the context of a small Forth program leads to answering the research question.

1.4 Aim of the Work

As an integral part of this thesis, a prototype for optional, pluggable typing in the context of Forth is implemented in the functional programming language Haskell. This implementation is crucial in the process of evaluating the *practical* use of the type system since limiting the language's expressiveness is inevitable in light of the undecidable nature of static type checking. Such restrictions, including the trade-offs of type inference and type annotations, can be best assessed with a working prototype in order to derive *pragmatic* solutions useful for a Forth programmer.

The targeted Forth subset to reason about in the prototype includes the single-cell types of the 1994 Forth ANSI standard and the according words of the CORE wordset. The following checks and language features are integrated into the static analysis:

- Type consistency of the composition of stack effects
- Type consistency of colon definitions
- Compliance of a colon definition with an optional type annotation
- Loops and control structures
- Multiple stack effects
- Reference types
- Subtyping
- Compile-Time Programming
- Object-Oriented Programming
- Higher-Order Programming

Orthogonally, the trade-offs between *type inference* and *type annotations* (and stack effect comments, respectively) must be analyzed in the above contexts in order to derive a *pragmatic* solution.

In addition, the prototype is designed to support the creation of checkers of varying static type safety which should offer a customizable trade-off between static type safety and expressiveness to the user. An evaluation of their use given a small Forth program will lead to fully answering the research question.

1.5 Structure of the Work

This thesis starts with a survey on the dichotomy of static and dynamic typing in Chapter 2. Their defining properties are specified and terms related to type systems are clarified for later use.

Chapter 3 puts the focus on state-of-the-art attempts which aim to blur the strict divisioning of exclusively static, respectively dynamic, type systems in favour of more integrated type system – in the process, terms as *gradual typing*, *soft typing*, *optional* and *pluggable* types will be clarified. Those findings motivate the use and choices made for the design of the prototype later on.

Chapter 4 will introduce the characteristics and idiosyncrasies of the Forth programming language focusing on its computational model, the stack, its susceptibility to static analysis and type checking.

Existing work to build a static type system for Forth will be revised in order to find a theoretic underpinning for an implementation of an optional, pluggable type system. The design and implementation of the Haskell prototype are discussed in Chapter 5.

In Chapter 6 the resulting type system's capabilities, limitations and restrictions must be evaluated. It will be compared to the examples of Chapter 3 and analyzed in the context of optional, pluggable typing. In order to demonstrate its practical use, the checker will be applied to a small Forth program under different typing configurations.

CHAPTER 2

Comparing Static and Dynamic Typing

2.1 The Nature of Static and Dynamic Typing

From a programmer's perspective, there is hardly any concept as self-evident as the notion of types. Across the majority of programming languages, it feels natural to associate a type like *Integer* to a value designating a number and thus tagging it with that type, transferring the human ability to classify the nature of a thing from the real world to the world of programs. Thus, a type denotes a set of values.

As common as types are in programming languages, they are not to be taken for granted, though. Apart from assembly languages or formal languages like the λ-calculus, even some high-level languages are *untyped*, e.g. Forth, where any function only expects its argument to be a sequence of bits without any further restriction. A program written in a typed language, however, can be *type checked*, i.e. for all operations it is checked whether a function is applied only to arguments of a compatible type such that the whole program is *type consistent* (also called *type safe* and *type correct*) or not. Type checking can be done at compile-time (also called static type checking) as well as at runtime. A program is type consistent if and only if the types of all expressions are compatible with the types expected by their context as implied by the rules of the type system.

Following the nomenclature of [CW85], strongly typed languages can guarantee type consistency at compile-time while weakly typed languages cannot. Those terms are often confused with the popular dichotomy of statically and dynamically typed languages; however, that distinction is about the point of time where a type can be assigned to a value: In contrast to dynamic typing, static typing allows for assigning a compile-time unique type to every expression. Thanks to those compile-time type assignments, statically typed languages are also strongly typed in practice. Conversely, there are still dynamically typed languages which are strongly typed, namely object-oriented languages,

which can guarantee compile-time type consistency by static type checking (see [JG97], p. 326).
While the concept of static type consistency is crucial in the context of the optional type system created in this thesis, the notion of unique compile-time type assignments is also relevant regarding type inference and type annotations (see Section 2.3). Therefore, it's the differences of the qualities of programming of statically and dynamically typed languages that are compared in the following section.

2.2 The Trade-offs of Static and Dynamic Typing

2.2.1 Benefits of Static Typing

Program verification. As type consistency can be decided at compile-time given static typing, a broad range of errors can be identified *earlier* than in weakly-typed dynamically-typed language: Whereas a type error would only be discovered at run-time using the latter, the same error would have stopped the compiler to translate the program in a statically-typed language in the first place. [KJS10] concludes that "static type checking is by far the most widely used verification technology today", thus increasing the *trust in a program's functional correctness.*

Early error detection. It is widely acknowledged that the cost of recovering from errors increases the later they are detected in the software engineering process (see [Bo81], [Wes02] and [SDD+04]) Eliminating type errors at compile-time, static type checking *fails cheaper* than dynamic type checking and has an economic advantage in that respect.

Optimization. In addition, statically-typed languages can perform better with respect to performance – as dynamic type checks are not necessary at runtime the compiled program can run *faster* (see [WC97]).

Documentation. Static types and type annotations respectively, provide some kind of documentation on the actual program code. Being less informative than natural language documentation from a semantic point of view, static types are checked formally, though, and are thereby much more authentic.

Refactoring and Evolution. In general, nowadays, programs need to be adapted and maintained for a long time after the initial writing. Any change in a program, however, runs the risk of introducing errors in a seemingly remote part of the code base. Static types reduce that risk as introduced type inconsistencies are noted immediately across the whole program code. Likewise, an integrated development editor can execute more sophisticated non-local refactorings in the context of static typing as feasible in the scope of dynamic typing. Actually, in [Unt12] a static type system is applied to programs of the dynamically typed language Smalltalk just for that sake of refactoring!

2.2.2 Drawbacks of Static Typing

Rejection of some safe programs. The benefits of static typing don't come without this serious caveat, often referred to as a lack of *expressiveness*. Its distinctive demand of asserting a compile-time unique type to every expression excludes a whole range of programs which are not well-typed in that specific sense but would still run without any type errors in every execution path.

In the below listing we see the pseudo-code of a function which either returns a string or a number by means of an if-statement. Is it possible to assign a compile-time return type to `stringOrInteger1` although its then-clause and else-clause are differently typed? Given a compiler sophisticated enough to erase the if-clause altogether this is certainly possible - the determined unique type is a string. It is not difficult to come up with a scenario where static analysis fails in that respect, though, no matter how sophisticated it is. In `stringOrInteger2` the truthiness of the condition depends on applying the function < on its arguments `5` and `7`; obviously, without explicit knowledge about the semantics of < the condition cannot be decided statically. Anyway, there is no sense in educating the type checker about < as it still wouldn't know how to deal with other unknown functions assigned to `myFunction`.

```
1: procedure stringOrInteger1
2:     if false then          ▷ Type checker could prove statically that false is not true
         return "hello"
3:     else
         return 5
4: procedure stringOrInteger2
5:     myFunction ← (<)       ▷ Type checker cannot prove statically anything about (<)
6:     if myFunction 5 7 then ▷ Static analysis cannot derive that the condition
         return "hello"                                  ▷ evaluates to true
7:     else
         return 5
```

As a consequence, sound and complete static type checking is *undecidable* in the general case. A sound, static type system always rejects some safe programs and is often referred to be *conservative* – in contrast to *optimistic* type checking which accepts programs that are not compile-time provable type consistent.

Verbosity. In order to check the consistency of a program's operations, information on the operations' types must be gained in the first place. Even in presence of a static type system which features some sort of type reconstruction, the programmer would need to state some types in certain cases by means of type annotations.

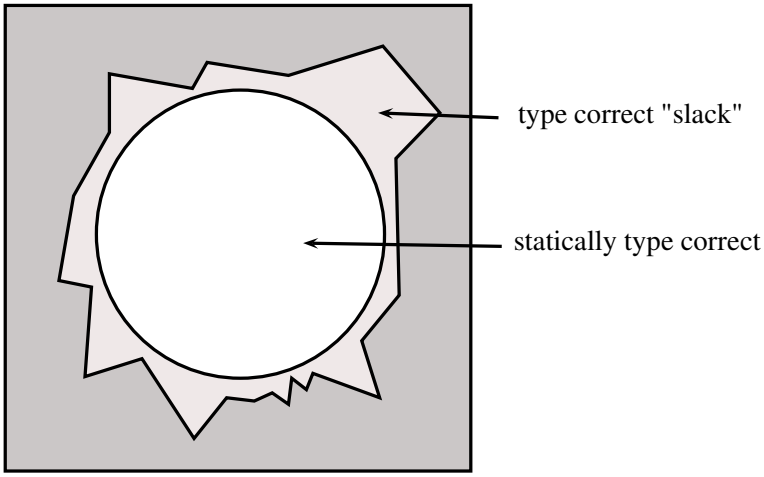

Figure 2.1: Static typing always excludes some type correct programs

2.3 Discussion

While the above mentioned benefits and drawbacks of static typing are partly based on formal reasoning (undecidability of sound and complete static type checking), measurable metrics (performance bonus given static optimization) or widely acknowledged software engineering experience (e.g. the benefits of early error detection and better refactoring support), some of them are disputable and considering them an advantage or a disadvantage is a matter of interpretation.

Program Verification

Critics of static typing claim that its 'program verification' aspect of static, and thus earlier, type error detection is not that significant *in practice*. In that line of thought, static typing provides a false sense of safety as the detection of static type errors does *not* rule out all unwanted runtime errors – for example, array bound checks still need to be done at runtime. Other practically useful guarantees as the non-emptiness of lists usually cannot be checked by a typical static type system, either. In fact, it is not uncommon that even the property of type consistency is deliberately violated in some constructs of static type systems and thereby the soundness of the system is spoiled altogether – see covariant arrays in Java as a popular example. As a consequence, one might argue that a drawback as static verbosity outweighs the minor benefit of more or less static type consistency, more so given an even known to be unsound type system. Anyway, even type systems claimed to be sound might not be applied correctly by the compiler. Cardelli [Car96] critisizes the common absence of precise type system specifications which would make possible the unambiguous translation and verification of type checking algorithms.

Expressiveness

Figure 2.1 graphically shows the relation of type consistent and statically provably type consistent programs. Static type checking always goes with that so-called 'slack', denoting the type correct programs of which their type consistency can not be shown at compile-time due to the aforementioned undecidability of general, static type checking. Referred to as a lack of expressivity by dynamic typing advocates, it's at the same time claimable that the slack mostly refers to language constructs and idioms which are not a good idea to use anyway.

Subtyping and Type Annotations

The discussion gets even more controversial if it extends to programming languages with support for subtyping and object-oriented programming, respectively. Gaining a lot of promotion with the rise of object-oriented programming, subtyping more generally refers to a rule of substitutivity: "If A is a subtype of B, then elements of type A can be used anywhere that an element of type B is required" ([Mit95], p. 1).

As to the verbosity induced by type annotations, statically typed languages without subtype polymorphism as the early ML and Haskell have done fairly well, given their type systems with Hindley-Milner type inference. *Type inference*, often called *type reconstruction*, originally refers to the ability of a type system to automically deduce the most general, so-called *principal type* (see [Jim96]) of an expression at compile-time. Hindley-Milner type inference and a respective implementation like *Algorithm W* [Gra06] provide automatic type reconstruction for all expressions given an unannotated program in the context of a type system with universal quantification over types. It has proved useful in minimizing type annotations although known to be undecidable in the light of polymorphic recursion. Generally, Hindley-Milner type inference gathers global equality constraints on related expressions such that their unification results in type assignments. This scheme does not work as effectively any more in the light of subtyping as the subtype relation introduces inequations, such that the problem gives rise to the undecidable semi-unification problem [KTU90].

Alternatively, *local* type inference infers "missing type information only from the types of adjacent syntax nodes" ([HP99], p. 1) so that global unification variables are avoided. However, *some* type annotations are needed as a consequence so that this approach is called *partial* type inference in contrast to *full* type inference.

Object-Oriented Programming

The introduction of subtyping has made type inference in its original sense even more unattainable, as the principal type property is impossible to fulfill [FM90]. Ignoring that property, types inferred in the presence of subtyping can get *impractically* big as they

```
class A
  int foo(x: int) = ...

class B
  int foo(x: int) = ...

bar obj = 1 + obj.foo(3)
```

Listing 2.2: What type should be inferred for 'bar'?

need to include type constraint sets as in [Mit91] or in [EST95] for an object-oriented setting.
Consequently, object-oriented languages cannot provide the same level of practical type reconstruction as languages with Hindley-Milner type inference in the general case; extra verbosity in form of type annotations is imperative for compile-time type checking.

It needs to be mentioned that the topics of object-oriented programming, type inference, subtyping, static and dynamic type checking are further intertwined as to notions such as dynamic dispatch, static type based overloading, class-based or prototype-based object-orientation, single inheritance or multiple inheritance, encapsulation, structural and nominal subtyping, bounded quantification etc. As an example, Listing 2.2 shows the difficulty of type inference when there are multiple classes having a method of the same signature. What type should be inferred for `bar` and its argument, respectively? No most general type can be inferred if the respective language only permits nominal subtyping. Supporting duck typing or union types (see Listing 3.2) would be imperative to express the most general constraint that `obj` has to provide a method `foo` with an according signature.

A full analysis on the interplay of the above concepts is highly language specific and beyond the scope of this thesis. However, related design decisions are mentioned in the integration of object-oriented programming in Section 5.14.

Program Evolution and Agility

There is a clash of arguments to the question whether static typing or dynamic typing does better in the field of program *evolution* and *agility* in particular. As mentioned before, static types make sure that a small change in a large code base does not introduce type inconsistencies in a far-off part of the program, however, this forces the programmer to actually update all affected remote code and the corresponding type declarations in order to compile the program. In the context of Web software engineering such expectedly longer cycles induced by that latter inconvenience tend to outweigh the benefits of global type consistency; obviously, the dynamically-typed scripting language Ruby has enjoyed great popularity as a server-side scripting language in web applications as conciseness

and *agility* have proved more important than type safety in the web engineering context. Similarly, in the contexts of rapid prototyping or interactive programming with a demand of high dynamicity, dynamically-typed languages like Python are prevalent.

The Versatility of Types

In general, we conclude that the question whether static or dynamic typing is better to use can only be answered in the context of specific scenarios and languages since external criteria as safety criticality, performance constraints, the given software engineering context and process, the expected size of a code base or the cost of errors can cast a different light on their characteristics.
Above all, however, typed languages themselves already differ significantly from each other as to how much can actually be expressed by types in the respective language. Apart from canonical *type checking*, a simple classification of values into sets and the verification of the correct use of operations on them, types influence optimizations, automatic documentation, program structuring or can even entail computations on the type-level [KJS10]. Consequently, it is deceptive to draw conclusions about the practicality of static typing judging from a specific statically-typed language with its type-specific feature set. In [MD04] Meijer et al. address the underlying problem what language features programmers associate with static or dynamic typing and what typing-related language features they actually expect to use, respectively, e.g. subtype polymorphism which, historically, has become popular in the context of object-oriented programming in statically-typed languages. Furthermore, there are mentioned generics, (unsafe) array covariance, lazy evaluation, prototype inheritance, higher-order functions and, above all, contracts and type inference. By contracts Meijer et al. on the one hand refer to the promise of static type consistency which they consider a contract too weak to be useful in practice, on the other hand, more useful contracts typically express invariants which need to be monitored at run-time, conversely. Type inference is claimed to be indispensable by Meijer et al. as it's a nuisance to force programmers to write type declarations although the compiler could easily infer the types.

Given the analysis above that static and dynamic typing each work best in specific contexts and Meijer et al.'s analysis on what programmers actually want when they long for static or dynamic typing, the benefit of a combined typing approach becomes obvious. Integrating static and dynamic typing should provide the best of two worlds. Meijer et al. express the motto "Static typing where possible, dynamic typing where needed!" and Chapter 3 deals with implementations of this approach.

CHAPTER 3

Combining Static and Dynamic Typing

The idea of combining static and dynamic typing dates back to the 1990s and has been omnipresent ever since. In recent years, it has found its way into mainstream programming as last but not least the JVM and the .NET platform, originally hosting statically-typed languages, have opened their doors for dynamic features and dynamic languages, respectively; from C# 4.0 on there is the keyword *dynamic* for declaring references which are not statically type checked by the compiler but at run-time only. In addition, *Dart* and *TypeScript*, referred to as either optionally or gradually typed, have been established by the tech leaders Google and Microsoft.
In the following an overview of different approaches of blurring the line between statically and dynamically typed languages will be given. Exemplarily, some implementation details will be briefly sketched to demonstrate the difficulties of incorporating languages features as subtyping into the specific approach.

3.1 Soft typing

Chronologically, soft typing is one of the earliest approaches, introduced by [CF91] in 1991. Cartwright and Fagan layer a static type system on top of the core of the ML language without subtyping. It's main idea reads as follows: given a program in a dynamically-typed language, a static type checker does not need to reject operations which can't be shown to be statically type consistent but it suffices to insert run-time checks in those places instead. Conversely, run-time checks can be eliminated if they can be proved to be unnecessary at compile-time. As a result, soft typing claims to retain the advantage of early error detection of static typing while allowing for increased expressiveness. However, in order to be a practical, useful solution it is necessary to find a way to satisfy the two conflicting goals that Cartwright et al. postulate: the type system

```
include? : (Fixnum) -> Boolean
include? : (String) -> Boolean
```

Listing 3.1: List each conjunct of an intersection on its own line in [FAFH09]

"must be rich enough to type 'most' program components written in a dynamic typing style without insertig any run-time checks, yet simple enough to accomodate *automatic type assignment*" (often called *type inference* or *type reconstruction*). In particular, the need to accept *unannotated* programs is essential to retain a program's dynamic character. Cartwright et al. provide an implementation building on union types, recursive types and circular unification in such a way that they can reuse the same Hindley-Milner polymorphic type inference algorithm as standard ML. Thus, they intend to satisfy the above criteria and create a *practical* soft type system.
Cartwright and Wright have refined that work in [WC97] where they build a soft type system for a Scheme dialect. They build a soft type system which eliminates 90% of the run-time checks necessary without soft typing. Moreover, due to static optimization their programs run 10-15% faster than their dynamically typed equivalents according to their benchmarks.

In addition, soft typing has been applied to languages as diverse as PHP [CHH09], Erlang [Nys03] and Ruby [FAFH09]. Obviously, the problems faced in soft typing the diverse constructs of those languages differ immensely. For instance, [FAFH09] uses so-called *intersection types* to model the prevalent feature of ad-hoc overloading of functions in dynamic languages like Ruby.

Consequently, `include?` of Listing 3.1 has *both* the types `(Fixnum) -> Boolean` and `(String) -> Boolean` such that static type checking of `include?` succeeds if its argument is of type `Fixnum` *or* `String` and the result has the type `Boolean`. Conversely, [FAFH09] uses union types as in Listing 3.2 to give a static type to a variable upon which a method is called which exists in two classes. Anyway, the concept of inferring union types and intersection types has been popular throughout diverse approaches for inferring static types.

Generally, it is worth mentioning that in favor of practical usefulness *annotations* have been introduced in one way or another: [FAFH09] features type annotations of which

```
class A; def f () end end
class B; def f () end end
x = ( if ... then A.new else B.new)
x.f
```

Listing 3.2: [FAFH09] infers that the static type of x is 'A or B'

the correctness is taken for granted at compile-time and only checked at run-time. In addition, those soft type systems partly exclude whole language constructs as classes and prototypes in [CHH09] or can not guarantee inferring a static type for a method although it might have one [FAFH09]; basically, those type systems are not even close to being *sound* and generally need some kind of annotations for typing certain language constructs.

All in all, soft typing does well at improving the performance of the originally purely dynamically-typed code ([WC97]). However, in general, it does not fully live up to its original claim of integrating all the advantages of the dynamic and static typing worlds. Soft typing does not allow to take a crucial part of a program and make it reliably safe by enforcing a static typing constraint on that piece. This is where gradual typing can provide better support.

3.2 Gradual Typing

The underlying idea of gradual typing is being able to add static type information *incrementally* in such a way that compile-time typed and untyped values play together nicely. In [ST07] Siek and Taha aim for that smooth interaction "by allowing the programmer to control whether a portion of the program is type checked at compile-time or run-time by adding or removing type annotations on variables". Obviously, type annotations are thereby crucial in gradual typing in order to tag portions of the code which should be type checked at compile-time. In addition, gradual typing guarantees static type consistency for code pieces which are *fully annotated*.
It has been in the area of gradual typing where there has been done profitable work on how to make subtyping work together with type consistency in the light of combining static and dynamic types.

Subtyping

Subtyping has been problematic to handle in the context of mixing static and dynamic typing from the beginning. To this end, it's generally necessary to put a "dynamic type" in relation to the known static types. It is in the early work of [ACPP89] that Abadi et al. introduce that canonical *Dynamic* type in order to integrate dynamic typing in a statically typed language; the seamless conversion of static and dynamic types is however not the case in that approach as explicit *injection* and *projection* operations are required. Thatte advances that approach of managing a *Dynamic* type together with static types in his so-called "Quasi-Static Typing" approach in [Tha90] . In the light of subtyping, Thatte puts *Dynamic* at the top of the subtype hierarchy (allowing up-casts) and at the bottom (allowing down-casts) at the same time in order to check the type consistency of operations on dynamic and static values. The transitive nature of the subtype relation, however, then implies that every single type is related to every other type.

Listing 3.3 would not result in a type error as $bool \leq Dynamic$ and $Dynamic \leq int$ imply that $bool \leq int$. As a consequence, wrongly used subtyping would not cause

```
def add1(x : int) -> int:
  return x + 1
add1(true)
```

Listing 3.3: Type checks under Thatte's subtype relation

any static type errors any more. In [ST06] Siek and Taha show that even in presence of Thatte's plausiblity checking, which should have excluded that inconsistency, the type system fails to catch many thereby induced errors. As a consequence, Siek and Thaha dodge that problem altogether in [ST07] as in their gradual typing approach the dynamic type is not at the top of the subtype hierarchy but rather *neutral* with regard to subtyping. In fact, gradual typing defines a consistency relation on types which is succintly summarized in [SV08], essentially giving up the transitivity of Thatte's subtyping relation. As a result, a type error as in Listing 3.4 is at least reported at runtime, in spite of the dynamic type.
Gradual typing inserts *wrappers* or *proxies* when untyped values are cast or coerced to typed values. These references make sure that at run-time all interactions with these values abide by the guarantees, claimed at compile-time.

```
x : Dyn = true
y : Int =  x
// Gradual Typing adds a cast and a runtime check as follows
x : Dyn = true
y : Int =<Int> true // Runtime Error!
```

Listing 3.4: A runtime check protects the subtype consistency relation [HTF10]

Listing 3.4 shows how gradual typing inserts a runtime check where a static type interfaces with the dynamic type `Dyn`. Those runtime proxies impose a certain runtime penalty and can cause unpleasant surprises like space leaks or losing tail recursion. Space-efficient solutions for those problems have been introduced by the work of [HTF10], though.

Blame Control

Gradual typing can be adapted to provide *blame control* as outlined in [SGT09]. The notion of *blame* actually stems from [FF02] where Findler and Felleisen introduce "blame" for breaking contracts for higher-order functions. As far as gradual typing is concerned, blame control is used to make sure that a fully annotated piece of code (of which the type consistency can be shown at compile-time according to gradual typing) does not get *blamed* for a run-time type error when it gets applied to an incompatible dynamic value;

instead, the place in the code where the incompatible value gets wrongly casted will be blamed! This mechanism is particularly useful when you need to interface with untyped third-party code. Although blame control induces further run-time effort to store and manage all casts a reference goes through, it improves the debugging of gradually typed programs anyway.

Gradual typing examples

There have been various publications on how to adopt the smooth migration of dynamic and static types of gradual typing to specific languages or language features, e.g. [SG12] illustrates the challenges to accept for creating efficient implementations. Supporting generics is outlined in [II09] and elaborated on in [II11], the notion of mutable objects in gradual typing is investigated in [SVB13] and type inference is probed in [SV08] as well as in [RCH12] where the gradually typed *ActionScript* is enhanced with partial type inference.

The most mentionable attempts to apply gradual typing to existing languages are *Gradualtalk* in [ACF$^+$13] for Smalltalk and *Typed Scheme* in [THF08] for Racket where *Typed Scheme* makes the programmer take the decision between static or dynamic typing on a per-module basis. In addition, *Typed Scheme* supports so-called *occurrence typing* (see [THF10]) where the type systems takes predicate checks into account, i.e. the type $((Number \cup Boolean) \rightarrow Boolean)$ is derived for the following expression using a runtime type check. Thereby, $(Number \cup Boolean)$ denotes a union type:

$$(\lambda\,(x : (Number \cup Boolean))\ (if\,(number?\,x)\,(=\,x\,1)\,(not\,x)))$$

As to *Gradualtalk*, it is worth mentioning that it supports blame control and gives rise to an optional type system (see Section 3.3) in case the runtime casts and checks are switched off.

3.3 Optional typing

Bracha [Bra04] introduces the idea of optional, pluggable type systems. Building on his work on the static type system Strongtalk for Smalltalk [BG93] Bracha dismisses the idea of *mandatory* typing, no matter if static or dynamic. In fact, he claims that "the dichotomy between statically typed and dynamically typed languages is false and counterproductive. The real dichotomy is between mandatory and optional type systems." In particular, it is the fact that so far mainstream programming languages have been intrinsically tied to their single type system by means of the language's syntax and semantics which is considered unnecessary and even harmful. As a replacement, Bracha proposes the use of optional, *pluggable* type systems such that multiple type systems can be run for diverse semantic analyses. There are two criteria which characterize such a type system:

1. An optional, pluggable type system "has no effect on the run-time semantics of the programming language"

2. An optional, pluggable type system "does not mandate type annotations in the syntax"

Obviously, the first criterium alone already has a strong impact on what language features a compatible language can have. Thus, Bracha rules out public fields, class based encapsulation and static type based overloading to appear in an accordingly designed language, claiming that those features are not to be recommended in the first place, anyway. In particular, the first requirement that embodies the concept of 'optionality' is regarded as more crucial than the second. The second requirement merely states that type annotations should not be mandatory parts of the language syntax but should be optional. In fact, Bracha does suggest using user-defined annotations, as they already exist in Java or C# for example, for annotating nodes of the abstract syntax tree in order to support multiple type systems; that discussion also touches upon the field of type inference of which the importance Bracha considers exaggerated in "soft" type systems. As a matter of fact, he proposes to make type inference just equally optional and to provide type inference algorithms as a service of the IDE.

Benefits

In general, optional, pluggable types should improve the language modularity (as the semantics do not depend on the type system any longer) and could prove beneficial in terms of program and language evolution (programs that are not accepted by a type checker now could be accepted by a different type checker later on while retaining the same runtime semantics).
A whole range of static analysis can be expressed as a type checking problem (Bracha mentions aliasing, ownership, information flow), however, it is hardly practical to add those analyzers directly to the language. This is where the idea of optional, pluggable types shines when they could be provided by means of a general *framework*. In that case exotic research projects providing some sort of static analysis would not languish in the unknown any longer but would have a place where they could be found and "plugged-in" by the programmer.

Optional typing examples

Aside from Bracha's original optional type system Strongtalk [BG93] for Smalltalk, optional typing has been applied in recent years, too, e.g. 'Typed Lua' [MMI14] and 'Typed Clojure' [BS12] add optional, static type systems to Lua and Clojure. Listing 3.5 shows how [BS12] uses annotations to give static types to heterogeneous maps, an inherently dynamic construct.. The type `person` is defined to only have the mandatory keys `:first-name` and `:last-name`, an optional numeric key `:age` and no other keys. As a result, the annotation in Line 8 fails as `michael` has an additional key `:middle-name` which is not allowed to occur in the `person` type.

```
1  (defalias person (HMap :mandatory {:first-name String
2                                      :last-name  String}
3                         :optional {:age Num} :complete? true))
4
5  (ann myself person)
6  (def myself {:first-name "Gregor" :last-name "Riegler" :age 26})
7
8  (ann michael person)
9  (def michael {:first-name "Michael" :middle-name "Max"
10               :last-name "Mustermann"})
```

Listing 3.5: Checking the types of heterogeneous maps in Typed Clojure

[LG11] provides a static type system for a Python-like language and offers so far unknown optional runtime monitoring of type errors. As throwing an exception in case of a type error would affect the runtime semantics and would thus break the defining constraint of optional typing, type errors are only logged in such a way that the program itself can not respond to it. So the runtime semantics remain unaffected while the logged type errors provide a good source for later debugging.

TypeScript is a language that compiles to JavaScript and it is generally referred to as gradual. As a matter of fact, types are erased on compilation and type errors may not be detected at compile-time nor at run-time. Consequently it differs noticeably from the original approach of Siek and Taha and is rather optionally typed. In particular, *TypeScript* is designed to be deliberately unsound in favor of pragmatic use; [RSF+14] proposes an alternative type system for *TypeScript* which maintains soundness while claiming to be practical at the same time.

Optional, pluggable typing examples

The above examples provide optional typing, however, they neglect the idea of "pluggable" types insofar as they do not provide a "framework" for creating or plugging in those type systems. In contrast, the TypePlug framework of [HDN09] for Smalltalk is designed with that idea in mind.

TypePlug

Creating type systems. TypePlug provides an interface for creating optional type systems for Smalltalk. Essentially, one needs to subclass the class `TPTypeSystem` and override some of its methods in order to define the types and the rules of the type system as well as its annotations. Concerning the types, it's worth mentioning that every thereby created type system explicitly contains a top type denoting the

unknown type; a value has that type if it is either unannotated or its type can not be derived by TypePlug's type inference. That top type is necessary to make partially annotated source code type check.

TPTypeSystem. Concerning the methods to override, `is:subtypeOf:` and `unifyType:with:` are essential as the former specifies the subtyping relation and latter defines the unification operation. The TypePlug framework then calls those methods as part of the general type checking routine. Generally, only (partially) annotated code is type checked, thus making the integration of third-party code easy.

Type annotations. TypePlug makes heavy use of annotations for specifying types. Upon subclassing `TPTypeSystem` it is thus necessary to declare a method `systemKey` which uniquely identifies annotations for that type system by prefixing every annotation. As a consequence, every expression can have more than one annotation, namely one per type system. Using *Persephone* [DDLM07] any node of the abstract syntax tree of the Smalltalk program can be annotated and thus be given a type. Those annotations can be either evaluated statically at compile time or at runtime through Smalltalk's reflection API.

Handling third-party code. The TypePlug framework provides a code browser which makes the definition of external annotations possible, i.e. annotations which are not part of the code. This is useful in case the source of standard library classes would need to be annotated otherwise, which would pose a practical problem in the context of packaging and distribution. Even more important, external annotations provide a way to integrate third-party code in the type checking procedure.
Cast annotations are another feature useful in that respect. They can be used to cast an expression to an arbitrary type and should be used at the interface to third-party code; casts are not checked but their type assertions are taken for granted in the type checking process.

Pluggable types for Java

In recent years, Java has seen the emergence of various approaches to make its type system more expressive – in this respect, an *optional* type system has the advantage not to interfere with the conventional Java toolchain, its compiler and runtime system as it takes action before those are used. Among those, the *Checker framework* is the practically most relevant approach. It makes heavy use of annotations for type specifications, which has been facilitated by JSR 308 that permits annotations to appear on any use of a type. Those additional type specifications are then checked for correctness by the various checkers it provides, e.g. a nullness checker, a lock checker, a format string checker and others. The *Checker framework* has been introduced and compared to other frameworks in [PACJ+08] and [DDES11]. Figure 3.3 shows an application of its nullness checker which recognizes a possible *Nullpointer* exception at line 5. Dereferencing `bar`, however,

is deduced to be safe thanks to the `@EnsuresNonNullIf` annotation and the checker's flow sensitive analysis.

```
1  public class Foo {
2    public void interact(@Nullable Foo bar, @Nullable Foo baz) {
3      // Safe dereference as bar cannot be null
4      if(equals(bar)) bar.doNothing();
5      baz.doNothing();    // Possible null dereference
6    }
7    @EnsuresNonNullIf(expression="#1", result=true)
8    @Override
9    public boolean equals(@Nullable Object obj) {
10     return obj != null;
11   }
12   private void doNothing() {}
13 }
```

Listing 3.6: Flow-Sensitive Nullness checker

JavaCOP [MME+10] is a framework that provides checkers similar to those provided by the Checker framework. In addition to annotations, however, it uses a domain-specific language for a declarative specification of typing rules expressing rules as contraints of the nodes of the abstract syntax tree. As a result, type system rules can be defined by the user. The following example rule thus expresses a nullness checker's important rule that a possible null value may not be assigned to a reference which is annotated with a `@Nonnull` annotation [jav10].

```
rule nonnullAssign(JCAssign a){
  where(hasNonnullAnnot(a.lhs)){
    require(defNonnull(a.rhs)):warning(a, "Possible assignment
    of null value to @Nonnull reference!");
  }
}
```

3.4 Discussion

This chapter has given both a broad theoretical overview as well as several examples of diverse approaches of enhancing a programming language's host type system: soft typing, gradual typing and optional, pluggable typing. Now the question remains what mentioned approach fits best this thesis' task of adding types to the Forth programming language.

In addition, it is worth contemplating if the above examples provide any features which are not tightly coupled to their particular typing task but could be adopted for a typed Forth development process.

Typing Forth

Forth is an untyped language – there is no runtime type checking done such that the soft typing and gradual typing approaches' defining quality of adding static types to an existing dynamic type system is not imperative. Optional, pluggable typing, however, fits best the challenge of deriving compile-time guarantees on the one hand and retaining runtime efficiency on the other hand.

The main verification task of this work's Forth optional typing prototype will be the correctness of the operations with regard to the types of values they take from the stack and return to the stack – much more high-level checkers as provided by the JavaCOP or the Checker framework are thus beyond the scope of this thesis. That's why there will not be developed a declarative typing rules language as used in JavaCOP; the pursued approach rather resembles TypePlug having a rather fixed type checking algorithm and providing various 'hooks' which actually qualify the resulting type system.

Adoptions

Virtually every mentioned approach uses *annotations* for type specifications, leveraging the existing annotation system in the case of Java or developing a new specification language as in some soft typing examples. Thus, it will be imperative to adopt an annotation language which integrates nicely into existing Forth code.

Support for the incremental adoption of more static type consistency besides compile-time untyped code can be adopted from gradual typing. That idea is made more precise in the scope of optional typing by the TypePlug system which only type checks partially annotated code by configuration.

Another feature worth adopting is TypePlug's support for *casts*. While the notion of a cast typically undermines any claim of verified correctness it is a useful feature for the incremental adoption of static types and could be activated optionally such that it does not circumvent any typing rules if it is not meant to be used.

CHAPTER 4

Forth Language Characteristics

In the following, the idiosyncrasies of the Forth programming language are briefly outlined, focusing on the importance of the stack and the notion of *compile state* and *interpret state*, *compilation semantics* and *interpretation semantics* respectively, which are crucial for an understanding of how Forth definitions are created and behave. Memory management and similar issues are neglected altogether as they do not matter in the scope of static program analysis. A more thorough language description is available as part of the ANSI Forth standard [ans94]. All code has been tested with the ANSI compliant Gforth, version 0.7.0.

Stacks. Forth is first and foremost a stack-based programming language. In contrast to other languages, a *stack* is not only used in contexts where first-in-first-out semantics are needed algorithmically, but in Forth the *data* stack provides the underlying computational context of execution, meaning that routines take their arguments from the stack and push their results onto the data stack, e.g. Forth's arithmetic operations read exactly like reverse polish notation: `"2 3 + 5 *"`. In addition to the data stack, the standard defines the return stack, the floating point stack and the control flow stack.

Words. There are no syntactic criteria limiting the juxtaposition of Forth operations; as a matter of fact, parsing Forth is rather simple compared to parsing languages of different paradigms. Obviously, given a line of Forth source code, the Forth text interpreter parses operation after operation, which is made easy as they are separated by whitespaces and thus called *words*, and in each case it executes the word's semantics, generally without the need for any look-ahead or look-back; even words for manipulating the control flow are handled the same way, making heavy use of the (control-flow) stack (see Figure 4.1). Due to that strict top-down processing user-defined words need to be defined before their first application in file order.

Creating words. The canonical way of creating a word is the so-called colon definition. Upon executing `":"` the well-known variable `State` is set where `State` can either be *compile state* or *interpret state* denoted by 0 or 1. Essentially, the colon thus enters compile state, meaning that the Forth text interpreter creates a definition `myplus` in the so-called *dictionary* (this is the piece of memory where user definitions are compiled to) and compiles (= appends the semantics to the current definition) the words to come into the body. The `";"`, conversely, finishes the current definition and enters interpret mode again.

```
: myplus ( n1|u1 n2|u2 -- n3|u3 ) + ;
3 5 myplus ( pushes 8 onto the data stack )

: constant create , does> @ ;
3 constant PI ( defines an integer constant for PI )

: circle-area ( n1 -- n2 ) dup * PI * ;
3 circle-area ( pushes 27 onto the data stack )

: ball ( n1 -- n2 ) dup dup * * [ PI 3 / ] * ;
```

Apart from `":"`, the words `create` (parses the next word from the input stream and creates an empty definition with that name) and `does>` (defines the interpretation semantics of the last word that was defined by means of `create`) and others as `variable` or `constant` are used for creating definitions.

Semantics. If `State` is in compile state, the *compilation semantics* of a word are executed, otherwise the *interpretation semantics* of a word are executed. In general, *interpretation semantics* denote the canonical effect of the operation, e.g. taking the first two values of the stack and returning the addition in case of "+", while *compilation semantics* denote appending the interpretation semantics to the definition which gets currently compiled.
This ambiguous nature of Forth words can be used to apply "compile-time" programming, i.e. executing the interpretation semantics in compile state, by words as `"["` and `"]"` which manipulate the `State` variable directly. However, there are words like `"If"` (see Figure 4.1) which do not have any interpretation semantics, i.e. they can't be used outside of a compiled definition, consequently, and the standard calls their behavior *run-time semantics* when they are run in the context of the containing compiled definition.

Stack effects. The stack effects of a word specify how it changes the stack, i.e. what values it takes from the stack and pushes onto it. In idiomatic Forth code the stack effect is supplied as part of a colon definition in the context of a Forth comment, e.g. ($before_2$ $before_1$ -- $after_2$ $after_1$), where $before_x$ specifies the values that the word consumes and $after_x$ specifies the values that get put on top of the stack; we later refer to the consumption and production types of a stack effect as

6.1.1700 IF

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: -- *orig*)
Put the location of a new unresolved forward reference *orig* onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until *orig* is resolved, e.g., by **THEN** or **ELSE**.

Run-time: (*x* --)
If all bits of *x* are zero, continue execution at the location specified by the resolution of *orig*.

See: **3.2.3.2 Control flow stack, 6.1.1310 ELSE, 6.1.2270 THEN.**

Figure 4.1: Description of the word "IF" in the ANSI Forth standard [ans94]

stack image. The right-most type literal corresponds to the top of the stack, e.g. $before_1$ and $after_1$ in the example above. It is a stack comment that is referred to whenever a *type annotation* or a *specification* of a word or a definition is mentioned in the following. Being mere comments, stack comments are considered optional in the standard – they are not checked for consistency in any way – still there are conventions to name the arguments in the 'before' and 'after' parts according to their intended types. In particular, *multiple* stack effects or multiple argument types are specified by separating the alternatives with "`|`".

Data types. In Section 3.1 of the ANSI standard [ans94] Forth's intended, but unchecked, data types are listed which are then used in the glossary to specify the stack effects of Forth's standard library's words. That listing comprises above all numerical types, flags, address types and system types for specifying the effects the text interpreter executes on entering definitions and dissolving control structures, respectively. Besides, those data types differ in the size they occupy on the stack – one or two cells. Moreover, a simple, transitive subtype relation on numerical and address types is outlined, e.g. `a-addr` (aligned address) is a subtype of `c-addr` (character-address) is a subtype of `addr` (address).

Input stream types. When a word reads one space-delimited word from the input stream, like `create`, this is expressed in a stack effect as `( "<spaces>name" -- )`. Apart from `<spaces>` there are other delimiters as `<paren>` and `<quote>` to express that the word read from input stream is delimited by parens or quotes. Such an input stream type is thus put in between quotes and displayed in the consuming stack image.

CHAPTER 5

Designing Optional, Pluggable Types for Forth

In this chapter the design and an according prototype of optional, pluggable types for Forth are outlined. The targeted Forth subset to reason about in the prototype includes the single-cell types of the 1994 Forth ANSI standard [ans94] and the according words of the CORE wordset. Starting with a Forth parser, integrating the CORE words and modelling stack effects, literature on static Forth program analysis is reviewed in order to find a theoretical basis for checking type consistency in the stack-based Forth context. After deriving an implementation for a consistent composition of simple stack effects, the following checks and language features are integrated into the static analysis:

- Type consistency of the composition of stack effects
- Type consistency of colon definitions
- Compliance of a colon definition with an optional type annotation
- Multiple stack effects
- Reference types
- Subtyping
- Compile-Time Programming
- Object-Oriented Programming
- Higher-Order Programming

The above language features are integrated in a way which allows for guaranteeing the compile-time type consistency of a program. Orthogonally, there will be outlined implementation details of the system's configuration ability of giving rise to those optional checkers of varying degrees of static type consistency that are presented in Chapter 6, e.g. additional language features like casts, assertions and the use of the `Dynamic` type. Those configuration options will then be summarized in Section 5.16.

Implementation Language

The implementation is done in the functional programming language Haskell. Why isn't it done in Forth, given its role of type checking Forth code? This is a deliberate choice – as an optional type system (see Section 3.3) should have no effect on the runtime semantics of the programming language, the elsewhere reasonable choice of keeping all options (e.g. for using dynamic, runtime type checking) by integrating the type checking in the Forth interpreter loop does not prove necessary. On the contrary, using a different language for that static analysis emphasizes that stringent quality of an optional type system and makes sure the compliance of that rule. In addition, Haskell's algebraic data types, static typing, pureness and thereby better adherence to formal specifications than common in most languages are considered beneficial in the task of crafting a prototype of a type system.

5.1 Integrating CORE Forth words

Before designing and implementing a stack effect calculus, a method of integrating the words of the Forth CORE wordset and their associated stack effects needs to be defined. More precisely, both the COMPILESTATE as well as the INTERPRETSTATE stack effects must be derived as well as the information whether the word changes that state. All that information is fetched from [ans94]. In order to simplify that process of defining those words, a simple domain specific language is created with the help of the Free Monad Pattern [CK14] to finally get at values of type `Word` as in Listing 5.1. The values of type `CompilationSemantics` and `InterpretationSemantics`, if defined, contain the respective stack effects.

Stack Notation

It is convenient to take the prevalent stack notation for defining the stack effects and the type signatures of Forth words, respectively. As it is necessary to parse stack effects later on when user-given type signatures need to be verified, we already use that parser for processing the effect strings in Listing 5.1.

Data Types. Compared to the stack effect notation of [ans94] we restrict the accepted type literals to those single-cell types which do not correspond to system types:

```
data Word = Word {
    parsed          :: String
  , compilation     :: CompilationSemantics
  , interpretation  :: InterpretationSemantics
  }

colon :: Word
colon = do
  parsing  ":"
  entering COMPILESTATE

plus :: Word
plus = do
  parsing "+"
  effect  "( n n -- n & u u -- u )"

questionDup = do
  parsing "?dup"
  effect "( x -- x / x -- x x )"
```

Listing 5.1: Modelling the CORE forth words

flag = boolean flag, *char* = character, n = signed number, $+n$ = non-negative number, u = unsigned number, x = unspecified cell. Those data types will be called primitive types below. Optionally, a numeric index can follow the type literal, which is needed for supporting type variables in Section 5.10. A number literal, e.g. `5`, has the type n. A type literal for reference types is introduced in Section 5.11.

Input Stream Types. We restrict the allowed delimiters of input stream types (see Chapter 4) to white-space delimiters. An input stream type's type literal consists of a string surrounded by quotes. Furthermore, we make a more precise distinction as to the meaning of input stream types than in the Forth standard, syntactically: Either the word read from the input stream should name a newly created variable or it does not. On the one hand, the stack effect of `create`, which creates a new variable, reads `( :"name" -- )` where `name` can be an arbitrary string. Thus, the colon marks an input stream type which causes the creation of a variable. In contrast, the effect of the word `char`, which reads a string from the input stream and outputs its first character, reads `( "name" -- char )`. Either way, an input stream type can occur only in the consuming stack image.

Multiple Effects. We introduce different encodings for defining alternative stack effects, which can be seen in the stack effects of `plus` and `?dup` in Listing 5.1.

In the standard, the effect of `plus` reads `( n1|u1 n2|u2 -- n3|u3 )` which could be instantiated to `( n1 n2 -- n3 )` or `( u1 u2 -- u3 )`, however, that description could be interpreted as `( n1 u2 -- n3 )` as well because the symbol `|` designates a stack type alternative only locally. Anyway, addition (and other words operating on numbers) should have *both* the types `( n1 n2 -- n3)` and `( u1 u2 -- u3 )`. In contrast, the definition of `?dup` sees its multiple stack effects joined with the slash character (having *either* the first *or* the second effect) which has implications on type checking (see Section 5.5).

5.2 Parsing

The **Parsec** parser generator library has been chosen for the job of parsing the above mentioned stack effects as well as tokenizing and parsing the input programs. Parsec offers a parser monad transformer interface in order to integrate other effectful operations as logging, changing the parsing state, reading from a configuration and error management.

Lexical Analysis

First of all, the input program string is transformed to a list of tokens. A token is either a Forth core word that was integrated as seen in Listing 5.1 or an unknown identifier. In the next processing phase, that unknown identifier could turn out to signify an input stream parameter or a user-defined word.

Building an Abstract Syntax Tree

In this parsing phase we build Parsec parsers which operate on the list of tokens stemming from the lexical analysis. Those tokens are grouped into values of the type `ForthWord` or `Expr` (see Listing 5.2), modelled as alternatives of the `Either a b` type alias `Node`. We thus build an abstract syntax tree representation of the input program.
The `CheckerState` is used in that process to maintain information on the current `State` value, the known core words and the gathered user-defined words. In addition, two lists of stack effects for `COMPILESTATE` and `INTERPRETSTATE` stack effects are maintained. Extensions in future sections will add other information to that global state, e.g. Section 5.6.2 adds the stack effects of colon definitions and Section 5.11.3 adds the names and types of created variables to that state.
The type checking process will be kicked off as soon as another `Node` has been parsed.

In Listing 5.3 the fundamental architecture of the input programm processing is illustrated. `parseNode` delegates to the parsers `expression` and `evalForthWord`. The parser `expression` succeeds if and only if one its alternatives separated by the `<|>` operator succeeds. Using the example of `parseIfElse`, it is worth mentioning that we gain the AST by treating tokens as `if`, `else` and `then` solely by their intended syntactic meaning – in contrast to the Forth runtime interpreter which actually treats all words equally and compiles such a control structure by pushing branch addresses onto the control-flow stack and resolving them, respectively.

```
type Token = Either Unknown Word
type Node = Either ForthWord Expr

data CheckerState = CheckerState {
    definedWords      :: M.Map String Definition
  , coreWords         :: M.Map Parsable Word
  , stateVar          :: SemState
  , isCompiling       :: Bool
  , compileEffects    :: [StackEffect]
  , interpretEffects  :: [StackEffect] }

data ForthWord = UnknownE Unknown
         | UserWord (CompiledOrExecuted (Name, [StackEffect]))
         | CoreWord ParsedWord

data Expr = IfExpr [Node]
       | IfElseExpr [Node] [Node]
       | ColonExpr String (Maybe Semantics) [Node]
       | DoLoop [Node]
       -- ... other expressions
```

Listing 5.2: Parsing the input program

On failure of all `expression` parsers, however, the `evalForthWord` parser is tried; that parser only succeeds if the input token corresponds to a core word or corresponds to a word defined before in the program. It returns an according value of type `ForthWord` which contains the stack effects of that word, the information whether it changes the `State` variable and in addition, input stream arguments are resolved. At last, `check` (see Listing 5.6) is called with that parsed value.

5.3 Forth Static Analysis

Stack-based languages and Forth have experienced the increased interest of the static analysis community in the early 1990s. The works of Pöial ([Pö90], [Poi91]) and Stoddart [SK93] and Knaggs [Kna93] introduce rules for composing stack effects for the purpose of calculating the accumulated effect from any number of input words. That act of stack effect inference is used as a synonym for *type inference* in the following. Given a small number of rules it is possible to infer a stack effect from the composition of two stack effects which are already *known*. That general composition approach does not only work for core library words (of which the effects are known anyway) but also scales to more practical programs with colon definitions given by the user: The definition of the colon definition must always precede its first application in Forth due to the workings of the interpreter. As a result, Forth stack effect inference gets by with local type inference and

```
parseProgram :: CheckerM [Node]
parseProgram =  many parseNode

parseNode:: CheckerM Node
parseNode = do
  forthWordOrExpr <-       liftM (new _Expr)       expression
                   <|> liftM (new _ForthWord) evalForthWord
  check forthWordOrExpr
  return forthWordOrExpr

expression :: CheckerM Expr
expression = try parseIfElse <|> try parseIf
             <|> parseColon -- and other expressions

-- A sample expression parser
parseIfElse :: CheckerM Expr
parseIfElse = do
  parseWord W.'if'
  ifExprs   <- manyWordsTill W.'else'
  elseExprs <- manyWordsTill W.'then'
  return $ IfElseExpr ifExprs elseExprs
```

Listing 5.3: The parsing phase at a glance

is thus less complex than general type inference (see Section 2.3) where the resolution of global type constraints is the rule.

Notation. In Figure 5.4 the stack effect composition rules of a simple type system without subtypes are given. s_1, s_2, t_1 and t_2 refer to a *stack image*, denoting zero, one or more data types present on the stack. x and y refer to a single data type. The dot $\cdot$ concatenates stack images or single types where the right-hand side operand is on top of the stack and the arrow $\rightarrow$ denotes the transition of one stack image to another. $\#$ is a unary function expecting a stack image and gives the number of types contained in that stack image. The empty set $\emptyset$ denotes a type clash.

$$
\begin{aligned}
&1. \quad \#s_2 = 0 \implies (s_1 \rightarrow s_2)\ (t_1 \rightarrow t_2) = (t_1 \cdot s_1 \rightarrow t_2) \\
&2. \quad \#t_1 = 0 \implies (s_1 \rightarrow s_2)\ (t_1 \rightarrow t_2) = (s_1 \rightarrow s_2 \cdot t_2) \\
&3. \quad x \neq y \implies (s_1 \rightarrow s_2 \cdot x)\ (t_1 \cdot y \rightarrow t_2) = \emptyset \\
&4. \quad (s_1 \rightarrow s_2 \cdot x)\ (t_1 \cdot x \rightarrow t_2) = (s_1 \rightarrow s_2)\ (t_1 \rightarrow t_2)
\end{aligned}
$$

Figure 5.4: Composition rules for simple stack effects in [SK93]

Those rules are also present in [Kna93] where Rules 1, 2 and 3 are referred to as 'composition rules' and Rule 4 is named a 'reduction rule' as the size of the stack images decreases in that process.
More recently, Pöial has pursued an experimental Java implementation of static Forth type checking in [Pö08], based on the above preliminary approaches and his own groundwork in [Pö03] and [Pö06]. While the thereby introduced typing rules don't differ from those used by Knaggs and Stoddart, Pöial makes use of the concept of the *greatest lower bound* of two stack effects to deal with type checking the effect of an IFELSE control structure.

In [Kna93] and [SK93] respectively, Knaggs and Stoddart additionally outline integrating control structures, subtyping, reference types and polymorphism in the type system. Those changes can be done atop the simple composition rules of Figure 5.4 and allow for tackling the typing challenges as defined in Section 1.4. This is why the stack effect composition rules of [Kna93] and [SK93] will be taken as the theoretical underpinning of the type inference and the type consistency checking in the implementation.

5.4 Implementing Stack Effect Inference

The rules of Figure 5.4 form the base of the hereby introduced simple type checking approach. They support neither polymorphism nor subtyping. Those will be added in the sections 5.10 and 5.9.

```
data TypeSymbol = Flag | Char | N | Plus_N | U --- ... other types
data DataType = Wildcard | Primitive TypeSymbol
type IndexedStackType = (DataType, Int)

data StackEffect = StackEffect {
    before     :: [IndexedStackType]
  , streamArgs :: [DefiningOrNot]
  , after      :: [IndexedStackType] }

instance HasStackEffects Expr where
  getStackEffects (IfExpr forthWordsOrExprs) = do
    withEmptyState $ do
      effects <- mapM check forthWordsOrExprs

      let allEffects = (emptySt : effects) &
                       traverse.before %~ ( flag : )
      forbidMultipleEffects <- views allowMultipleEffects not
      when (forbidMultipleEffects && not (areTheSame allEffects))
         (throwing _IfExprNotStatic ())
      return allEffects
--... other expressions
```

Listing 5.5: Type Class HasStackEffects

5.4.1 Modelling Stack Effects

The type `StackEffect` has three fields: `before` refers to the stack image before the effect has been executed, `after` refers to the stack image after the execution of the effect and the list `streamArgs` denotes the arguments that are read from the input stream. As a Forth word can have multiple effects it is a crucial question whether that indeterminism is maintained in the checking process or if a single stack effect needs to be guaranteed at all times. This also has implications on how an IFELSE expression has to be dealt with: a unique stack effect can only result if the effects of the `if`-branch and the `else`-branch coincide. That treatment of `If` expressions needs to be enforced in a static typing approach. In favor of flexibility, the core algorithm can deal with multiple stack effects, though. As a result, the general enforcement of a single stack effect is a configuration option available in customizing the optional checker as described in Chapter 6. In this case, the matching of alternative branches is enforced if the respective configuration option `allowMultipleEffects` evaluates to `True` as can be seen in `getStackEffects` in Listing 5.5.

```
1  check :: Node -> CheckerM ()
2  check node = do
3    newEffects <- either getStackEffects getStackEffects node
4    currentEffs <- getCurrentEffects
5    let effects = [(eff1, eff2) | eff1 <- currentEffs,
6                                  eff2 <- newEffects]
7    reducedEffects <- mapM applyRule4 effects
8
9    let resultingEffects :: [MaybeT StackEffectM StackEffect]
10       resultingEffects = map applyRules1To3 reducedEffects
11
12   validEffects <- lift . fmap catMaybes . mapM runMaybeT $
13                   resultingEffects
14
15   let typeChecks = not . null $ validEffects
16
17   when (not typeChecks) $
18     throwing _Clash ()
19   setCurrentEffects validEffects
20
21 applyRules1To3 (stE1, stE2) = fmap fromJust . runMaybeT . msum .
22                       map ($ (stE1,stE2)) $
23                       [applyRule1, applyRule2, applyRule3]
```

Listing 5.6: Simple Type Checking outline

A type class `HasStackEffects` is defined of which the instances support a function `getStackEffects` which returns a list of `StackEffect` values. `Expr` and `ForthWord` are instances of that type class. Across the listings 5.6 and 5.5 the basic

type consistency check calling scheme looks like that: when `check` is called with an `IfExpr` as an argument the `getStackEffects` instance for an `IfExpr` first calls `check` on the Forth words of its body, couples those effects with the empty stack effect and prepends the consumption of a `Flag` type in both alternative effects. Thus, the stack effects of an expression are derived recursively from its constituents. In the case of loop expressions it is checked that the loop body leaves the stack untouched or produces a `flag` value in case of a BEGINUNTIL expression (see Section 6.1).

```
applyRule1 (stE1,stE2) = do
 guard $ null $ stE1 ^. after
 let before2 = stE2 ^. before
     after2 = stE2 ^. after
     streamArgs2 = stE2 ^. streamArgs
 return $ stE1 &~ before %= (++ before2)
       &~ after .= after2 &~ streamArgs %= (++ streamArgs2)

applyRule2 (stE1,stE2) = do
 let before2 = stE2 ^. before
     after2 = stE2 ^. after
     streamArgs2 = stE2 ^. streamArgs
 guard $ null before2
 return $ stE1 &~ after %= (after2 ++)
       &~ streamArgs %= (++ streamArgs2)

-- isSubtypeOf replaces matchesDataTypeExactly in Section 5.9
applyRule3 (stE1, stE2) = do
 (topOfEff1,_) <- hoistMaybe $ preview (after.traverse) stE1
 (topOfEff2,_)  <- hoistMaybe $ preview (before.traverse) stE2
 let typesMatch = topOfEff1 `matchesDataTypeExactly` topOfEff2
 guard typesMatch
 typeClash

applyRule4 stE1 stE2 = unpack $ applyRule4' stE1 stE2
  where
  unpack = fmap (^?! _Left) . runEitherT
  applyRule4' stE1 stE2 = do
   (topOfEff1,_) <- firstOf (after.traverse) stE1 ?? (stE1, stE2)
   (topOfEff2,_) <- firstOf (before.traverse) stE2 ?? (stE1, stE2)
   dataType <- hoistEither . note (stE1, stE2) $
               topOfEff1 `matchesDataTypeExactly` topOfEff2
   applyRule4' (stE1 & after %~ tail) (stE2 & before %~ tail)
```

Listing 5.7: Implementation of the composition rules of Figure 5.4

5.4.2 Implementing the Stack Effect Calculus

In Listing 5.6 it is illustrated how the rules of the theoretic stack effect calculus of Figure 5.4 are called to derive a type checking result. The function `check` first computes all pairs of the present stack effects and the effects of the next Forth expression. Those pairs are then processed by Rule 4: A stack effect pair is shortened as long as the top production stack type of the old stack effect coincides with the top consumption stack type of the new stack effect. Just then it is checked which one of Rules 1 to 3 is applicable and the stack effects pair is either reduced to one or a type clash is signaled. Rule 1 and Rule 2 not only compose the data types on the stack but also input stream types (see Listing 5.7).
If the resulting list of `validEffects` is not empty the current `Node` has been successfully checked for type consistency.

5.5 Handling of Multiple Stack Effects

In the context of the current expression having multiple stack effects, the above algorithm abides by the approach of [SK93] as it derives a type clash only if the list `validEffects` is empty, i.e. if no combination of the present stack effects and the stack effects of the current `Node` results in a type consistent composition.

```
\ key has effect ( -- char), hold has effect ( char -- )
: word1 if 4 else key then ; \ => ( flag -- n / flag -- char )
: word2 word1 hold ; \ => ( flag -- )
: word3 + > ; \ => ( n n n -- flag )
```

Listing 5.8: Composition examples with expressions having multiple stack effects

Listing 5.8 shows some colon definition examples. Their stack effects inferred by the presented algorithm, which implements the handling of multiple effects as designed in [SK93], are listed next to it.
The IFELSE expression in `word1` results in two stack effects inferred for that definition. Therefore, in Figure 5.9, checking the type consistency of `word2` results in checking two pairs of stack effects of which one type clashes by Rule 3 of Figure 5.4.

$$(flag \rightarrow n)\ (char \rightarrow) \Rightarrow \emptyset \qquad \textit{by Rule}\ 3$$
$$(flag \rightarrow char)\ (char \rightarrow) \Rightarrow (flag \rightarrow) \qquad \textit{by Rules}\ 4\ \textit{and}\ 2$$

Figure 5.9: Checking compositions of "if 4 else key then hold" in word2

The fact that `word2` type checks does not go with the claim to *guarantee* type consistency, however. It rather means that `word2` *may* have the effect ($flag \rightarrow$). There is an obvious solution for guaranteeing type consistency in all possible program runs:

Demand a valid composition of *all* stack effect pairs subject to the composition algorithm. Consequently, `word2` would not type check.

Using that approach, however, `word3` of Listing 5.8 would not type check either according to Figure 5.10, given that + has the effects ($n \cdot n \to n$) and ($u \cdot u \to u$) and > has the single effect ($n \cdot n \to flag$). Such a composition *should* work, however, given that the stack effect multiplicity of + has a different *meaning*.

$$(u \cdot u \to u)\ (n \cdot n \to flag) \Rightarrow \emptyset \qquad \textit{by Rule}\ 3$$
$$(n \cdot n \to n)\ (n \cdot n \to flag) \Rightarrow (n \cdot n \cdot n \to flag) \qquad \textit{by Rules}\ 4\ \textit{and}\ 1$$

Figure 5.10: Checking compositions of "+ >" in word3

Obviously, there are two kinds of multiplicity which need to be handled differently: On the one hand, *indeterministic effects* induced by a branching expression like IFELSE. On the other hand, *intersection types* like in the case of addition (see Section 3.1 and [Pie91], p.20). Stoddart and Knaggs themselves give an example of an intersection type in [SK93], page 10, where they define the word `and` to be the intersection of ($flag \cdot flag \to flag$) and ($logical \cdot logical \to logical$) which we write succinctly as ($flag \cdot flag \to flag \ \& \ logical \cdot logical \to logical$). Thus, they express that `and` should be applicable to `flags` as well as to bitwise logical entities, in general. Whereas their approach leads to the loss of the above depicted guarantee of type consistency in all program runs, this problem can be solved as follows.

Implementation. In addition to the list of stack effects of the stack effect composition algorithm's state (see Listing 5.2) the `CheckerState` gets a boolean field `intersection` which indicates whether those stack effects denote an intersection type. Furthermore, the function `getStackEffects`, which had been designed to return a list of stack effects `[StackEffect]` of the next `Node` to compose in Listing 5.5, also returns an according boolean value. Thus, an IFELSE expression always returns a `False` value in addition to the respective stack effects because it can never denote an intersection type. We call that boolean value `hasNodeIntersectionType` below. Given that additional information it now remains to define when the composition should be invalid as for `word2` of Figure 5.9 and valid for `word3` of Figure 5.10, i.e. when should *any* clash cause the failure of the whole composition?

The whole composition should fail if and only if at least one stack effect composition fails and both `intersection` and `hasNodeIntersectionType` are both `False`. On the one hand, the composition in Figure 5.9 fails as neither `word1` nor `hold` possess an intersection type. On the other hand, the composition in Figure 5.10 succeeds as + has an intersection type (see Listing 5.1). After a successful composition, the field `intersection` is set to `False` if the list of `validEffects` (see Line 12 of Listing 5.6) only contains one stack effect.

Forbid Multiple Effects. The above approach ensures the statically type consistent composition of operations having an indeterministic stack effect. Still, it can be claimed that a program *should* have deterministic stack effects only, thus following a best-practice leading to simpler, less complex programs. That's why the prototype supports the option `allowMultipleEffects` in the configuration of a respective checker. Listing 5.5 shows the prompt detection of the violation of the property in a branching expression. More generally, the property is checked when the computed, possibly indeterministic, stack effect of a colon definition would be exported into the `CheckerState`.

5.6 Type Checking Colon Definitions

5.6.1 Stack Effect Comments

In idiomatic Forth code it is common to add a *stack effect comment* at the beginning of a colon definition. That optional stack effect comment needs to be placed right after the name of the word that is defined. Using that idiom, the intended effect of a user-defined word is documented. Whereas in other languages such a type annotation is generally necessary to help type inference, this is not the case for Forth stack effect inference (see sections 5.3 and 2.3). However, it is practically useful to implement a static check whether the inferred stack effect of a word 'matches' the intended stack effect specified by the user. In addition, it is thus possible to restrict the use of a word in a type consistent way. Given that it matches the inferred effect, the effect of the stack effect comment is later used in the remaining type checking process whenever that word's effect is needed.

```
: add3 ( n -- n ) 3 + ;
: maybeDouble ( n flag -- n ) if 2 * then ; \ CLASH
: forcedDefinition ( n flag -!- n ) if 2 * then ;
: right2Effs ( flag -- n / flag -- char ) if 4 else key then ;
: wrong2Effs ( flag -- n & flag -- char ) if 4 else key then ; \ CLASH
: nAddition ( n -- n n ) + ;
: addition ( n -- n n & u u -- u ) + ;
```

Listing 5.11: Checking Stack Effect comments

In case of the colon definition of `add3` in Listing 5.11 it is trivial to decide whether the stack effect matches the inferred type – they are simply equal. In general, this task turns out to be harder, though, especially in the context of multiple stack effects. For single stack effects only, the approach of [SK93] is extended to also cover input stream types.

5.6.2 Matching Stack Effect Comments

Let *infI*, *specI* denote stack images and let *infE* and *specE* denote single stack effects in the following.

Case 1. In case a single stack effect *infE* is inferred, it matches the stack effect comment *specE* having the structure $(specI_1 \rightarrow specI_2)$ if and only if

$$(\rightarrow specI_1)\ \ infE\ \ (specI_2 \rightarrow) \Rightarrow (\rightarrow)$$

Notably, in the above composition, input stream types, which can normally only occur in the consuming stack image, can for once occur in the producing stack image of the stack effect $(\rightarrow specI_1)$. In that case, they are treated the same way as normal data types in the context of the composition rules of Figure 5.4.

Notably, *infE* is a subtype of *specE* in so far as, given the changes of Section 5.9 to stack effect composition, this rule respects contravariance in the data types of the consuming stack image and covariance in the data types of the producing stack image (see Section 5.14.2 for an example).

The above rule suffices to show that the implementation of `add3` matches with its stack effect comment. The rules for multiple stack effects will refer to that single-effect rule as a predicate *Matches*(*inferred*, *specification*) which is true when the single stack effect *inferred* matches the specified single stack effect *specification*.

Case 2. In case an effect comprised of multiple effects

$$(infE_1\ \ \&\ \ infE_2\ \ \&\ \ ...\ \ \&\ \ infE_k)$$

denoting an intersection type is inferred, it matches the stack effect comment

$$(specE_1\ \ \&\ \ specE_2\ \ \&\ \ ...\ \ \&\ \ specE_n)$$

if and only if there exists an injection $g : \{specE_1, ..., specE_n\} \longrightarrow \{infE_1, ..., infE_k\}$ such that

$$\forall x \in \{specE_1, ..., specE_n\}\ \ Matches(g(x), x)$$

Due to that rule the implementations of the words `nAddition` and `addition` of Listing 5.11 match their stack effect comments. It is because of that injective mapping that the context where an intersection type can be used can be safely reduced, as stated by the stack effect comment of `nAddition` of Listing 5.11.

Case 3. In case an effect comprised of multiple effects

$$(infE_1\ \ /\ \ infE_2\ \ /\ \ ...\ \ /\ \ infE_k)$$

which do not denote an intersection type is inferred, it matches the stack effect comment

$$(specE_1\ \ /\ \ specE_2\ \ /\ \ ...\ \ /\ \ specE_n)$$

if and only if there exists a bijection $f : \{specE_1, ..., specE_n\} \longrightarrow \{infE_1, ..., infE_k\}$ such that

$$\forall x \in \{specE_1, ..., specE_n\}\ \ Matches(f(x), x)$$

The stack effect specification of `maybeDouble` of Listing 5.11 reads less effects than the number of its inferred stack effects, consequently, there cannot exist a bijection. For `right2Effs`, however, there exists a bijection such that all mappings satisify the *Matches* predicate. `wrong2Effs` shows that the inferred effects never match the specified effects if either denotes an intersection type while the other does not.

Local Failure. In absence of a stack effect comment, the comprising words of a colon definition are composed in order to yield an accumulated stack effect which is then referred to when the colon definition's word is needed in later in the type checking process. When the composition algorithm yields a type clash in the body of a colon definition the whole type checking process of the program halts with an error. However, *practically*, it can be useful to tolerate a type clashing colon definition and continue type checking the rest of the program which does not make use of that erroneous definition. That configuration option will be called `allowLocalFailure` and will be used in Chapter 6.

Forced Effects. Listing 5.11 shows another feature supported by configuration in the stack effect comment of `forcedDefinition`, where `-!-`, delimiting the consuming and the producing stack image, denotes a forced stack effect. A forced stack effect bypasses static type consistency in so far as it specifies the stack effect of the annotated colon definition without matching it with the inferred stack effect. Thus, it's possible to use statically unsound or even unknown operations in an accordingly marked colon definition if this option is enabled.

Third Party Words. A real-world program often needs to access the words of a third-party library which cannot be accessed during type checking. That's why the checker supports the integration of third-party words by defining their stack effects as part of the initial checker configuration.

5.7 Assertions

When the computed stack effect of a word does not correspond to the specified stack effect, assertions can be helpful in the debugging process: an assertion is a comment of a certain kind which can be inserted anywhere in the body of a colon definition. An assertion specifies the expected stack image at that exact point of execution in the surrounding word. Assertions are a tool for the developer to specify more exactly the intended effects throughout the word's execution and can thus provide better error messages as to why a word does not correspond to its specification, respectively.

Syntactically, an assertion differs from an unprocessed comment by prefixing the list of expected types with `Assert` as in `( Assert n char )`; multiple stack effects are not supported. Internally, such a parsed assertion is trivially transformed to a regular stack effect such that it can be processed by the aforementioned basic stack effect composition algorithm and reduction rules:

`( Assert n char )` ⇒ `( n char -- n char )`

In Listing 5.12 another variant, a *strict* assertion is demonstrated, too; it is distinguished from a simple assertion by preceding the stack image with "`Assert!`". While a simple assertion also type checks asserting only the top part of the stack (Line 5) and supports an assumption as to the stack image before calling `dropNumber` (Line 1) a strict assertion forbids those cases; a strict assertion type checks only if the claimed stack image corresponds to the complete stack image computed from the effects of the preceding words of the surrounding colon definition!

```
1 : dropNumber ( n -- )        ( Assert  n ) drop ;
2 : dropNumber ( n -- )        ( Assert! n ) drop ; \ CLASH
3 : dropNumber ( -- )      5   ( Assert  n ) drop ;
4 : dropNumber ( -- )      5   ( Assert! n ) drop ;
5 : dropNumber ( -- n ) 5 4    ( Assert  n ) drop ;
6 : dropNumber ( -- n ) 5 4    ( Assert! n ) drop ; \ CLASH
```

Listing 5.12: Assertions

In addition, assertions play a crucial role in supporting higher-order programming in Section 5.15.

5.8 Casts

Similar to assertions, a cast can be specified by inserting a comment at the appropriate place, e.g. `( Cast n -- flag )` could be frequently used to cast the top stack value of type `n` to a boolean `flag`. Casts are thus a tool useful in the typically iterative process of adding more static types to an existing untyped program as they allow for avoiding a strict typing routine at selective spots – this obviously comes at the cost of breaking the default static typing, that's why casts must be allowed by configuration (see Section 6.1 for an example).

Concerning the implementation, casts are handled in a way similar to assertions: a comment starting with `Cast` is parsed to be a cast such that a word with the intended effect is generated and passed to the base stack effect composition algorithm. Effectively, the generated cast stack effect can thus evoke a type clash if the algorithm does not derive the cast's consuming stack image at the spot of its occurrence.

5.9 Subtyping

[SK93] proposes modifying the last two composition rules of Figure 5.4 in order to support subtyping. Figure 5.13 shows those simple modifications where $x \leq y$ means that `x` is a subtype of `y`:

Those modifications imply that a successful composition of stack effects does not require their top stack types to be equal but to be in a subtype relation. Conversely, a type clash results whenever that subtype relation does not hold in Rule 3.

3. $\neg(x \leq y) \implies (s_1 \to s_2 \cdot x)\ (t_1 \cdot y \to t_2) = \emptyset$

4. $(x \leq y) \implies (s_1 \to s_2 \cdot x)\ (t_1 \cdot y \to t_2) = (s_1 \to s_2)\ \ (t_1 \to t_2)$

Figure 5.13: Subtyping support of composition rules of [SK93]

Implementation. Subtyping can be integrated into the stack effect calculus implementation without a fundamental break of the design. In fact, only the calls of `matchesDataTypeExactly` in Listing 5.7 are replaced by `isSubtypeOf` as defined in Listing 5.15. There it is checked whether the first argument $t1$ is a subtype of $t2$ by checking their membership in a set specifying the subtype relation, a set of the `CheckerState`. That set was precomputed before the start of type checking. Effectively, the direct subtypes of primitive data types need to be specified as part of the configuration of the checker (see `checker4` in Listing 6.2) so that the transitive closure of the unfolding relation gets stored in that global `CheckerState`.

5.10 Introducing Polymorphism

There are a number of Forth words which only manipulate the order of the values on the stack, no matter what type those values hold. Using the example of `swap` with the effect `( x1 x2 -- x2 x1 )`, there are the type variables `x1` and `x2` which should be instantiable to any concrete type. Below, type variables are referred to as *wildcards*, following the nomenclature of [Kna93]. In stack effect comments a wildcard is always depicted by the type symbol `x` and the same optional index can make wildcard types unify within that stack effect comment. In fact, every data type can have an optional type index in the following. Whenever no optional index is given, a new index gets assigned. That new index is unique in the scope of the stack effect, e.g. `( n x1 -- x1 n )` gives rise to `( n2 x1 -- x1 n3 )`.

As a matter of fact, we need to extend the above stack calculus to support wildcard types. In [Kna93] so-called 'wildcard rules' are specified for this purpose. They need to be applied before the implementation of the stack effect composition rules in order to reduce the stack effects solely with respect to wildcard types. There are four such wildcard rules and Figure 5.14 shows how each rule applies to an example stack effects composition. `[x/y]` designates a substitution, i.e. type `x` replaces `y` in the scope of the enclosing parentheses. The arrow $\overset{Wx}{\Longrightarrow}$ means that the respective derivation is made according to wildcard rule x.

Wildcard Rules. Applications 1 and 2 of Figure 5.14 show the instantiation of the wildcard type with a concrete type if exactly one of the two top stack types to unify is a wildcard type. Example 3 demonstrates renaming a wildcard if the same wildcard index appears in both effects. The rule corresponding to Example 4, however, is applicable if the types to unify are both wildcards such that one wildcard type replaces the other.

1 $(flag \to n)(x_1 \cdot x_2 \to x_2 \cdot x_1) \overset{W1}{\Longrightarrow} (flag \to)\big((x_1 \to x_2 \cdot x_1)[n/x_2]\big) = (flag \to)(x_1 \to n \cdot x_1)$

2 $(x_1 \cdot x_2 \to x_2 \cdot x_1)(n \to flag) \overset{W2}{\Longrightarrow} \big((x_1 \cdot x_2 \to x_2)[n/x_1]\big)(\to flag) = (n \cdot x_2 \to x_2)(\to flag)$

3 $(x_2 \to x_2)(x_2 \cdot x_2 \to x_2) \overset{W3}{\Longrightarrow} (x_2 \to x_2)\big((x_2 \cdot x_2 \to x_2)[x_3/x_2]\big) = (x_2 \to x_2)(x_3 \cdot x_3 \to x_3)$

4 $(x_1 \cdot x_2 \to x_1)(x_3 \to x_3) \overset{W4}{\Longrightarrow} (x_1 \cdot x_2 \to x_1)\big((x_3 \to x_3)[x_1/x_3]\big) = (x_1 \cdot x_2 \to x_1)(x_1 \to x_1)$

Figure 5.14: Application of wildcard rules of [Kna93]

In the next section those rules will be adapted for being able to deal with reference types.

Implementation. Where do the wildcard rules fit into the original stack effect composition algorithm? The wildcard rule for renaming common wildcards is done just before the call of the stack effect reduction rule of `applyRule4` in Line 7 of Listing 5.6. In fact, the other wildcard rules and `applyRule4` then need to be applied in an alternating way to the resulting stack effects until a fixed point is reached. Just then `applyRules1To3` is applied.

5.11 Reference Types

As Forth is a stack-based language the concept of *variables* does not play a role as significant as in many other programming languages. Still, there are words as `create`, `variable` or `constant` in the Forth core wordset for creating named references to a value, where the reference's name is read from the input stream. Obviously, the way a type system deals with references has strong implications on what it guarantees: using static typing the type of a reference may never change and it must be inferred at compile-time.
The canonical Forth words for working with references are `!`, named "store", having the stack effect `( x a-addr -- )` and `@`, named "fetch", having the stack effect `( a-addr -- x )`. Obviously, a Forth reference is just a value of type `a-addr`, a memory address, which can be manipulated by arithmetic operations, such that direct, low-level memory management is the Forth default. In order to make references susceptible to static analysis, though, the concept of pointers is needed. To this end, *reference types* are introduced.

5.11.1 Notation

As [Kna93] proposes, we write `⋆k` in order to refer to a reference to a type `k` in stack effect comments. More generally, the number of stars prefixing a type `k` denotes the *reference degree*, the levels of indirections of the reference, while we refer to `k` as the *base*

type of the reference, e.g. `**n` has the reference degree 2, the base type `n` and can be written more succinctly as $\star^2$`n`. Consequently, the stack effects of "store" and "fetch" correspond to `( x *x -- )` and `( *x -- x )`. If the reference degree is 0, e.g. $\star^0$`n`, we get the primitive type `n`.

5.11.2 Adjusting the Wildcard Rules

The wildcard rules of [Kna93] need to be extended to deal with references to concrete values as well as with references to wildcards. Those amendments are detailled in [SK93]. Concerning the applications 1-4 in Figure 5.14, those changes with respect to the top stack types to unify have the following consequences during type checking.
Let the arrow $\stackrel{Wx}{\Longrightarrow}$ refer to the application of the respective wildcard rule x, let the arrow $\Rightarrow$ refer to an application of the stack effect composition of Figure 5.4 and let the type k designate a non-wildcard type in those examples:

ad 1) If the reference degree m of k is smaller than the reference degree n of the wildcard type, then there is a type clash.

$m < n:\ \ (\rightarrow *^m k\,)(*^n x \rightarrow) \stackrel{W1}{\Longrightarrow} \emptyset,\ \ e.g.: (\rightarrow n\,)(*x \rightarrow x\,) \stackrel{W1}{\Longrightarrow} \emptyset$

Otherwise the wildcard type is substituted with $*^{m-n}k$:

$$m \not< n:\ \ (\rightarrow *^m k\,)(*^n x \rightarrow) \stackrel{W1}{\Longrightarrow} (\rightarrow *^m k\,)\big((*^n x \rightarrow)[*^{m-n}k/x]\big) \Rightarrow (\rightarrow)$$
$$e.g.:\ \ (\rightarrow *n\,)(*x \rightarrow x\,) \stackrel{W1}{\Longrightarrow} (\rightarrow *n\,)(*n \rightarrow n\,) \Rightarrow (\rightarrow)$$

ad 2) If the reference degree m of k is smaller than the reference degree n of the wildcard type, then there is a type clash.

$m < n:\ \ (\rightarrow *^n x)(*^m k \rightarrow) \stackrel{W2}{\Longrightarrow} \emptyset,\ \ e.g.: (\rightarrow *x\,)(\,k \rightarrow) \stackrel{W2}{\Longrightarrow} \emptyset$

Otherwise the wildcard type is substituted with $*^{m-n}k$:

$$m \not< n:\ \ (\rightarrow *^n x)(*^m k \rightarrow) \stackrel{W2}{\Longrightarrow} \big((\rightarrow *^n x)[*^{m-n}k/x]\big)(*^m k \rightarrow) \Rightarrow (\rightarrow)$$
$$e.g.:\ \ (\rightarrow *x\,)(*\,*\,k \rightarrow) \stackrel{W2}{\Longrightarrow} (\rightarrow *\,*\,k\,)(*\,*\,k \rightarrow) \Rightarrow (\rightarrow)$$

ad 3) Wildcard renaming stays unchanged.

ad 4) Both top types are wildcard types.

$$n \le m:\ \ (\rightarrow *^m x_1)(*^n x_2 \rightarrow) \stackrel{W4}{\Longrightarrow} (\rightarrow *^m x_1)\big((*^n x_2 \rightarrow)[*^{m-n}x_1/x_2]\big) \Rightarrow (\rightarrow)$$
$$e.g.:\ \ (\rightarrow *\,*\,x_1\,)(*x_2 \rightarrow) \stackrel{W4}{\Longrightarrow} (\rightarrow *\,*\,x_1\,)(*\,*\,x_1 \rightarrow) \Rightarrow (\rightarrow)$$
$$n > m:\ \ (\rightarrow *^m x_1)(*^n x_2 \rightarrow) \stackrel{W4}{\Longrightarrow} \big((\rightarrow *^m x_1)[*^{n-m}x_2/x_1]\big)(*^n x_2 \rightarrow) \Rightarrow (\rightarrow)$$
$$e.g.:\ \ (\rightarrow *x_1\,)(*\,*\,x_2 \rightarrow) \stackrel{W4}{\Longrightarrow} (\rightarrow *\,*\,x_2\,)(*\,*\,x_2 \rightarrow) \Rightarrow (\rightarrow)$$

Briefly, the above modifications provide two enhancements compared to the wildcard rules of Figure 5.14: Firstly, they ensure the correct substitutions with respect to the reference degree of the substitutor, secondly, they signal a type clash when a wildcard type and a primitive type with incompatible reference degrees should be unified as in **1)** and **2)**. In the implementation, we have a clearer distinction of responsibilities concerning the reporting of a type clash between the wildcard rules and the stack effect composition rules. The wildcard rules only conduct the substitutions. A possible type clash resulting from incompatible reference degrees is detected in the stack effect composition rules of Figure 5.13 in the function `isSubtypeOf` anyway (see Listing 5.15).

```
isSubtypeOf :: T.DataType -> T.DataType -> CheckerM Bool
isSubtypeOf t1 t2 = do
  subtypeRelationSet <- view subtypeRelation <$> getState
  if refDegree t1 == refDegree t2 then do
    let baseType1 = baseType' t1
        baseType2 = baseType' t2

        inSubtypeRelation = fromMaybe False $ do
          type1 <- baseType1 ^? _Primitive
          type2 <- baseType2 ^? _Primitive
          return $ S.member (type1, type2) subtypeRelationSet
    return $ inSubtypeRelation || (subtypeByWcard baseType1 baseType2)

  else return False
  where
    subtypeByWcard b1 b2 = (has _Wildcard b1 && has _Wildcard b2) ||
      (not (has _Wildcard b1) && has _Wildcard b2) ||
      (has _Dynamic b1 || has _Dynamic b2)
```

Listing 5.15: Subtyping stack types

The presented approach already allows for checking the type consistency of polymorphic stack effects with respect to reference types. Still some questions remain: How does a reference get created in the first place and what primitive base type does it refer to? Can that base type be inferred or does it need to be specified by an annotation?

5.11.3 Creating References

Forth has the word `create` which reads the name of the variable to create from the input stream. We introduce various ways to define the type of a reference and to infer the type of a reference, respectively.

In the above figure various ways of defining and using the reference `foo` are depicted; in all those cases, the type of the reference `foo` is inferred to be `n`, a number.
The first line corresponds to initializing the reference on definition – the word `,` stores the top stack value in the lastly defined reference so that the reference can be trivially assigned the type of that value.

```
1 create foo 9 ,
2 create foo foo @ 4 +
3 create foo ( other words ) 9 foo !
4
5 : createNumberRef ( :"name":[ n ] -!- ) create ;
6 createNumberRef foo foo c@ \ CLASH
```

Listing 5.16: Exemplary Reference Usage

In Line 2 the type of `foo` stays unknown until the execution of `+` as that word expects two numeric values on top of which one stems from derefencing `foo`.
Line 3 shows another typical case as the type of `foo` is inferred at the time a value is written into that reference. Line 4 sees a colon definition with a forced stack effect comment. It contains an input stream type with an additional type annotation, giving the base type of the created reference. Any reference created with `createNumberRef` will then refer to a value of type `n`. As a consequence, there is a type clash in the next line as `c@` is a word which accepts its argument reference to contain a value of type `char`: Rule 3 of Figure 5.4 is responsible for the clash of the composition $(\rightarrow *n)(*char \rightarrow char)$.

Inferring the Referenced Type

It is necessary to adapt the so far type checking to infer the reference types as in lines 2 and 3 of Listing 5.16. That's why we add an `UnknownType` branch to the `DataType` type in Listing 5.17.
Whenever `create` (or a word containing `create`) reads the name of the new variable to create from the input stream, we add that name to a list of managed references in the global `CheckerState`. If the base type was specified as in `createNumberRef` or results from an initialization as in Line 1 of Listing 5.16, nothing has to be inferred. Any misuse of that reference would then result in a type clash induced by the wildcard rules or the stack effect composition rules.
If the base type of the reference has to be inferred, it is assigned the `UnknownType` with a unique identifier and stored in the global `CheckerState`.

```
type Identifier = Int
data DataType = Dynamic | Wildcard | UnknownType Identifier
              | Primitive TypeSymbol | Reference DataType
```

Listing 5.17: The UnknownType data type

An `UnknownType` behaves as a `Wildcard` in the so far wildcard rules with the following exception: Whenever an `UnknownType` is unified with a `Wildcard` (or `Dynamic`) it is the `UnknownType` which gets propagated.
In addition, when a wildcard `UnknownType` is unified with a primitive type (as in the

wildcard rule applications **a)** and **b)** of Figure 5.14), we add a side-effect to the normal wildcard rule application: the so far unknown base type of the reference is replaced with that now known primitive type in the `CheckerState`. As a result, the type of the variable is known and storing a value of a different type into that variable would signal a type clash.

5.12 The Dynamic Type

Listing 5.17 introduces `Dynamic` as an additional data type; it stands for any type in so far as `Dynamic` unifies successfully with any other value of the `DataType` data type in the scope of the wildcard rules, exactly as `UnknownType`, but also in `isSubtypeOf` of Listing 5.15. As a type literal in stack effects, `dyn` refers to that dynamic type. As a result, two configuration options arise for the checker:

- **Use dynamic type.** If enabled, you can use the literal `dyn` in stack effect comments. Due to its treatment in `isSubtypeOf` there can never result a type clash from the use of a `Dynamic` type in so far as the Rule 4 of Figure 5.13 is always applicable. For example, `myvariable` in Listing 5.18 is annotated to reference a dynamic type, so `c@` can read a character from it because of the type consistent composition

  $$(\rightarrow *dyn\,)\,(\,*char \rightarrow char\,) \Rightarrow (\rightarrow char)$$

  `dynamic2` shows how the dynamic type can be mixed with primitive data types in the stack effect comment which is still valid according to the approach of Section 5.6.

- **Dynamic CORE words.** If this option is used, all data types in the effects of all CORE words are replaced with the `Dynamic` data type, e.g. the stack effect of `+` then translates to `( dyn dyn -- dyn & dyn dyn -- dyn )` internally. Consequently, it is possible to only check the number of values of stack images in stack effect comments. Matching the inferred effect of `dynamic3` with its specification in Listing 5.18, a stack overflow arises, in contrast to `dynamic4`. `dynamic5` shows how a stack underflow is detected in combination with a strict assertion.

```
: dynamic1 ( :"name":[ dyn ]  -- ) create 9 , ;
dynamic1 myvariable myvariable c@
: dynamic2 ( dyn n -- dyn & u dyn -- u ) +  ;
: dynamic3 ( dyn dyn -- )     +  ; \ CLASH
: dynamic4 ( dyn dyn -- dyn ) +  ;
: dynamic5 + ( Assert! dyn dyn ) ; \ CLASH
```

Listing 5.18: Using the dynamic type

5.13 Compile-time Programming

As detailled in Chapter 4 the stack effect of a Forth word in INTERPRETSTATE can be different from the stack effect in COMPILESTATE, e.g. within a colon definition the words inside of [and] are executed at the time of the word definition. The overall compile-time stack effects of a colon definition are constrained to leave no value on the stack behind and further constrained not to expect anything on the stack at the beginning of the colon definition.

```
: foo [ 3 + ] ; \ CLASH as + expects 2 arguments
: add5 ( n -- n ) [ 5 ] literal + ;
: endif postpone then ; immediate
: 3or5 if 3 else 5 endif ; \ endif is executed at compile-time
```

Listing 5.19: Compile-Time programming

All in all, two sets of stack effects need to be managed in the CheckerState to check both INTERPRETSTATE and COMPILESTATE stack effects. The getStackEffects implementation for a specific Expr can then set the isCompiling field of the state, e.g. setting it to False in case of [ ], such that the checking algorithm combines the new effects either with the existing COMPILESTATE or the INTERPRETSTATE effects. This approach, however, does not suffice if a Forth word has both a compile-time and a run-time effect, as the word literal. It has the compile-time effect (x --) and the run-time effect (-- x). It takes the top value from the stack at compile-time and puts the same value onto the stack at the time the containing word is executed (see Listing 5.19). Obviously, that compile-time unification substitution must be carried along to the run-time effect consistency check so that the same wildcard substitutions can be performed.

In order to support this, the implemented wildcard rules therefore expect an optional StackEffect argument when the compile-time stack consistency is checked; that argument represents the potential run-time effect and all substitutions that are performed on the compile-time stack effect are equally performed on that optional argument. From this it follows that unifying the type of 5 with the compile-time wildcard of literal causes the same substitution for the wildcard of the run-time effect in Figure 5.19.

Limitations

Although the use of POSTPONE and immediate colon definitions are generally supported, a postponed word must not occur in the context of interdeterministic stack effects, e.g. in the branches of an IF or IFELSE expression, as it must be statically known during the parsing phase what words a definition thus compiles into another definition. Apart from that, the semantics of POSTPONE and immediate can be easily integrated into the existing design.

5.14 Object-Oriented Programming

There have been diverse projects to extend Forth for supporting object-oriented programming (see [RP96]). Gforth actually includes three object-oriented packages in its standard library. It has been decided to include support for the simplest of those, **mini-oof**, which adds class-based object-oriented programming with single inheritance and nominal subtyping. Parsers for the syntax of class definitions and method implementations are written in the sense of Listing 5.3 and the `getStackEffects` implementations of the resulting `Expr` execute the static analysis.

```
object class
   cell var age
   method likes ( object person1 -- person1 flag )
end-class person

person class
   cell var registrationNumber ( n )
   method study
end-class student
```

Listing 5.20: Class definitions and method implementations

The names of classes can be used in stack effect comments to denote a value of that type. The existing subtype relation on primitive types in the `CheckerState` is complemented with the subtype relation resulting from derived classes. There exists a supertype `Object` of all class types in the **mini-oof** library.

Figure 5.20 shows an example demonstrating the use of the object-oriented extension. Classes `Person` and `Student` are defined where `Student` is derived from `Person`. Type consistent implementations of the respective methods are given in Figure 5.21.

5.14.1 Methods

The hook for this type checking extension lies in the optional type annotations placed after the field and method declarations and after the `:noname` word in the method implementations. When a method declaration has a type annotation, the inferred effect of the implementation needs to conform to it, otherwise there is a type error. When a method is not annotated in the class definition the corresponding type signature is inferred from the implementation of the method. For instance, the stack effect `( a-addr u x1 -- x1 )` has been inferred from the words of the `:noname` definition of `study`. However, the correct effect for that class method is confined to `( a-addr u person1 -- person1 )` as all methods need to take the respective object as the top stack argument in the mini-oof library (this is necessary in order to access the given object in the method implementation as there is no *this* reference in that object-oriented extension). In the given example, the

top stack type `x1` of the consuming stack image is unified with `person`. Type checking would fail if the top stack type was not a wildcard or a data type other than `person`. Note that the data type `person` in the stack effect comment of the method `likes` in Listing 5.20 denotes the object upon which the method `likes` is called and does not stand for a method argument of type `person`. Thus, all data types of the consuming stack image except the top of stack are referred to when the *arguments* of a method are mentioned below.

5.14.2 Overriding Methods

```
:noname swap (Cast object -- person) age @ 40 > ; person defines likes
:noname [ person :: likes ] swap age @ 20 = or ; student defines likes
:noname s" I'm Studying! " type rot rot type ; student defines study
```

Listing 5.21: Type consistent method implementations

In Listing 5.21 the method `likes` of `Person` is overridden in the subclass `Student`. Using the mini-oof syntax for superclass method calls, `[ person :: likes ]`, the superclass method `likes` implementation is referred to.
The subclass `likes` method implementation has the inferred effect ($object \cdot student1 \rightarrow student1 \cdot flag$). The prototype executes a check whether this stack effect obeys the rules of contravariance in the method's data types of its arguments and covariance in its producing stack image with respect to the superclass method specification. For this purpose, we use the procedure defined in Section 5.6.2. As to the consuming stack images, only the method's arguments are taken into account. We derive the empty stack effect as required:

$$(\rightarrow object)\ (object \rightarrow student1 \cdot flag)\ (person1 \cdot flag \rightarrow) \Rightarrow (\rightarrow)$$

In contrast, the checker detects the attempt of wrong subtyping, e.g. when binary methods are attempted to be be overridden. This must not work as binary methods offend the constraint of contravariance of a method's arguments.

In Listing 5.22 the superclass specification for `equals` is ($point \cdot point \rightarrow flag$). That method should be overriden with an implementation having the signature ($betterpoint \cdot betterpoint \rightarrow flag$). This time we get the clashing composition:

$$(\rightarrow point)\ (betterpoint \rightarrow flag)\ (flag \rightarrow) \Rightarrow \emptyset$$

The composition fails according to Rule 3 of Figure 5.13 because $\neg(point \leq betterpoint)$. As a consequence, the checker refuses to accept that implementation of `equals` for `betterpoint`.

```
object class
   cell var location
   method equals ( point point -- flag )
end-class point

:noname location @ swap location @ = ; point defines equals

point class
   cell var size
end-class betterpoint

:noname ( betterpoint betterpoint -- flag ) size @ swap
          size @ =  ; betterpoint defines equals \ CLASH
```

Listing 5.22: Rejecting overriding a binary method

5.14.3 Fields

In contrast to methods, the data types which inherited fields refer to are checked to be invariant. The types of fields which are not annotated in their class definition can be inferred whenever they are used applying the same inference mechanisms as for references in Section 5.11.3.

```
person new age @ 12 * ;
```

Listing 5.23: Accessing a field of an object

In Figure 5.23 it is shown that fields of a class are accessed with `@` just as usual references – in this example, the type of the unannotated `age` of the class `person` is inferred to be `n`.

5.14.4 Inferring the Correct Class

The below listing makes use of the classes defined in Listing 5.20 and raises the question: What stack effect does the prototype choose for composition when the parser reads the word `age`? As `age` is a field of the classes `person` and `student` it could either compose ($student \rightarrow *n$) or ($person \rightarrow *n$). The call of `registrationNumber`, which is only defined for `student`, would however result in a type clash if the latter effect of `age` had been chosen. The type checking algorithm does not support any look-ahead, anyway.

```
dup age @ swap registrationNumber @ + \ has effect ( student -- n )
```

In this case, the prototype therefore inputs the intersection of all appropriate stack effects into the stack effect composition. Thus, the inferred effect reads

$$(\mathit{student}1 \to n \cdot \mathit{student}1 \ \& \ \mathit{person}1 \to n \cdot \mathit{person}1)$$

by the time `dup age @ swap` has been composed. That way, the composition with `registrationNumber` simply restricts that intersection in a valid way according to the rules of Section 5.6.2.

5.14.5 Limitations

- Only fields defined to be of size `cell` can be used. This is because only single-cell types are supported in the current implementation (see Section 6.2.1).
- There are no overloaded methods allowed to occur in a class. This is typical of a language supporting optional typing (see Section 3.3).
- Two classes which are not in a subtyping relation cannot define a method of the same name. Indeed, the second definition would just redefine the first and would thus make it inaccessible at runtime. That limitation avoids a potential stack effect inference problem as in Listing 2.2.
- If a class B extends a class A all method implementations of unannotated methods of A must be given above any `:noname` definitions which attempt to override superclass methods of A in B. This simply retains the top-down quality of the prototype's type checking process.

5.15 Higher-Order Programming

Typically, supporting higher-order programming is about being able to use functions as arguments and as return values. In Forth, an *execution token* represents such a function value on the stack. In order to integrate execution tokens into the type system design, it is necessary to deal with the Forth core words `tick` (which is parsed `'`) and `execute`.

`Tick` reads a word from the input stream and puts the corresponding execution token onto the stack; accordingly, the stack effect `( "<spaces>name" -- xt )` is specified in the standard. The word `execute` takes an execution token from the stack and applies the semantics which are denoted by it. In the standard, its stack effect reads as `( i*x xt -- j*x )` where `i*x` and `j*x` stand for any number of types of any kind as they are dependent on the effect denoted by `xt` . Furthermore, the standard mentions that in `INTERPRETSTATE` the following holds:

`' xyz EXECUTE` ≡ `xyz`

In order to reason about higher-order programming, it is necessary to retain that equality on the type level and in the stack effects of `tick` and `execute`, respectively.

In theory, that could be achieved by introducing type variables corresponding to `i*x`, which range over a sequence of types, using them to specify the effect of `xt` in the stack effects and extending the stack – however, the existing stack calculus rule system would need to be reconstructed heavily in order to support that unification of type variables with a sequence of types. That's why a less invasive way of reasoning over higher-order programming has been chosen.

We define the stack effect of `tick` to be

```
( "anyWord":[ EFFECT ] -- xt:[ EFFECT ] )
```

where `EFFECT` is a newly introduced concept of a type variable ranging over stack effects and may only occur in the scope of `:[ ]` which is allowed to occur for input stream types and execution tokens to specify their semantics. This obviously conveys the semantics of `tick` that the effect of the resulting execution token corresponds to the effect of the word read from the input stream.

In order to do without the above mentioned `i*x` type sequences variables, an application of `execute` needs a mandatory annotation as to the type of the execution token's semantics which it applies. That annotation is provided by means of an assertion (see Section 5.7) as illustrated in Listing 5.24: in `handleNumbers` the execution token stored in the variable `op` is fetched and gets executed after the mandatory assertion specifying its semantics.

```
: plus ( n n -- n ) + ;
: minus ( n n -- n ) - ;
create op
' plus op !
: handleNumbers op @ ( Assert xt:[ n n -- n ] ) execute ;
```

Listing 5.24: Higher-Order Programming

Limitations

In any case, there is a limitation as to the words which can be consumed by `tick`: they must be defined in the program or be included in the set of CORE words known to the checker, however, words with a purely syntactic meaning as `IF` or `:` cannot be used. This is because those syntactic words are only used for successfully parsing the abstract syntax tree and do not have stack effects on their own in the checker design.

5.16 Configuration Options

Throughout this chapter, several practical configuration options have emerged which influence the accomplished degree of static type consistency. These options can be enabled

or disabled irrespective of each other such that optional checkers with variable guarantees are created. Table 5.1 lists those options' names and their effect if the option is enabled.

Option	Description
Multiple effects	Allow the use of words which result in an indeterministic stack effect (Section 5.5).
Local failure	A type clash in a colon definition does not stop the whole type checking process. That option allows for type checking programs partially (Section 5.6.2).
Use `Dynamic` type	The `Dyn` type literal can be used in user stack effect comments (Section 5.12).
`Dynamic` CORE words	All data types mentioned in the stack effects of the provided CORE words are replaced with `Dynamic` (Section 5.12).
Forced effects	Specify a colon definition's stack effect that does not get checked against the inferred effect of the implementation (Section 5.6.2).
Casts	Cast the top stack type to a different type with a special stack effect comment (Section 5.8).
Mini-oof	Use the `mini-oof` extension for object-oriented Forth (Section 5.14).

Table 5.1: Boolean configuration options

In addition to the above boolean options, a subtype relation on primitive data types can be plugged in when the checker is created. Moreover, the checker can be fed a list of third-party words along with their unchecked stack effect specifications.

Subtypes	Provide a function of the type `TypeSymbol` ⇒ `[TypeSymbol]` with the intended meaning that the latter list value types are subtypes of the former primitive type.
Third-Party Words	Provide a list of the stack effect specifications of third-party words (Section 5.6.2).

Chapter 6 will evaluate the use of those options by applying different checkers to a small Forth program.

CHAPTER 6

Evaluation

The design and implementation of optional, pluggable types for Forth have been introduced in the previous section. Now the prototype gets evaluated as its practical use is demonstrated in the process of type checking a small Forth program. At last, the prototype is analyzed in the scope of optional, pluggable types and compared with related work.

6.1 Using pluggable Forth types

Listing 6.1 shows a small, functionally correct Forth program; it will now be type checked under different checker configurations to showcase the prototype's practical usage.

The example program implements a game often referred to as "Guess the number". Basically, you start the game calling `start-game` so that a random number between 0 and 99 is generated in `init-secret-number` and stored in the reference `secret-number`. Then, a loop repeatedly asks a number command-line input from the user and compares it with `secret-number`: If they match, the loop stops and the user wins; otherwise the user told is whether the searched-for number is smaller or larger and the loop continues.

First of all, it is worth mentioning that the program uses the `random` word and the `seed` reference included from the extension `random.fs`. While the current implementation does not check files which are included, it has support for bridging to third-party words as they can be added as part of the configuration. Alternatively, the `allowForcedEffects` and `allowLocalFailure` options can be used for defining a stack effect of a colon definition which uses unknown, imported words – the empty stack effect specified for `init-seed` is thus not checked in the checking process.

Listing 6.2 shows the checker configurations that are used in the following. Typically, let's begin with the weakest typing configuration `checker1`: by enabling `allCoreDynamic`

```
include random.fs
create secret-number

: create-random-nr 100 random ;
: init-seed ( -!- )
  utime drop here xor utime drop lshift seed ! ;
: init-secret-number
    init-seed create-random-nr secret-number ! ;
: read-guess
    pad 2 accept pad swap
    s>number? swap drop ;
: success cr ." You guessed it! " cr
    \ swap drop
    ;
: give-advice
    swap cr ." The requested number is "
    0 < if ." larger. "
        else ." smaller. " then cr ;

: feedback
    secret-number @ -
    dup 0= dup
    if
        success
    else
        give-advice
        ." Try again " cr
    then ;

: wrong-input
    drop ." Your input was not a number! " cr
    0 \ ( Cast n -- flag )
    ;

: start-game
    init-secret-number
    cr ." Guess the number between 0 and 99 " cr
    begin
        read-guess
        if feedback
        else wrong-input
        then
    until ;
```

Listing 6.1: "Guess a number" in Forth

```
checker1 = CheckerConfig {}
    & allowLocalFailure             .~ True
    & allCoreDynamic                .~ True
    & allowForcedEffects            .~ True
    & allowMultipleEffects          .~ True
    & thirdParty                    .~ [do { parsing "random";
                                             effect "( n -- n )" }]
    & allowCasts                    .~ False
    & allowOOP                      .~ False
    & allowDynamicType              .~ False
    & subtypes                      .~ (const [])

checker2 = checker1
    & allowMultipleEffects          .~ False
    & allCoreDynamic                .~ False
    & allowCasts                    .~ True

checker3 = checker2
    & allowMultipleEffects          .~ True

checker4 = checker2
   & allowCasts                     .~ False
   & allowLocalFailure              .~ False
   & subtypes .~ (\x ->
                   if | x == flag -> [ n ]
                      | True      -> [   ])
```

Listing 6.2: Creating different checkers

the types of all effects of all words are replaced with `Dynamic` such that no type clash but a stack underflow/overflow can arise.

In Figure 6.3 we can see part of the output of a run of the checker given that configuration. We see what types have been inferred for the definitions in the absence of any type annotations. All types being `Dynamic` or `flag`, there is an error in the type of `start-game` as the checker complains that the body of the loop does not produce exactly one `flag` value, which is a stringent requirement for a BEGINUNTIL loop. In order to get to the bottom of that problem, we look at the inferred types of `wrong-input` and `feedback`: It stands out that `feedback` has two stack effects and leaves one `flag` too many on the stack in one of them.

We continue disabling `allCoreDynamic` and set `allowMultipleEffects` to `False` such that the checker fails for `feedback` in Figure 6.4, consequently. The output gives away the inferred, different effects of the IF-ELSE branches. The source of error can thus be located more easily in the IF branch as an empty stack effect is inferred for `success` where the `flag` value is not removed from the stack – adding a `swap drop`

```
feedback
  ( dyn  --  Flag Flag )
  ( dyn  --  dyn )

start-game
 FAILURE: Body of BEGIN-UNTIL must produce exactly one flag value!

wrong-input ( dyn  --  dyn )
```

Figure 6.3: Detecting a stack overflow using the checker1

(see Line 13 in Listing 6.1) removes it and the problem disappears. The original Forth program unintentionally left a `flag` value on the stack in the success case – this did not impair the observable correctness of the program but can typically give rise to bugs which are hard to track down later on in a developing process.

```
feedback
 FAILURE: feedback: If-Else branches do not have matching types
  IF_BRANCH: ( Flag  --  )
  ELSE_BRANCH: ( N x1 Flag  --  x1 )

success (  --  )

give-advice ( N x1  --  x1 )
```

Figure 6.4: Part of the output running checker2

Using `checker3`, the checker now complains about a badly typed BEGINUNTIL expression again (see Figure 6.5), the reason being the numerical value it infers in one branch instead of a `flag` value.

```
start-game
 FAILURE: Body of BEGIN-UNTIL must produce exactly one flag value!
   (  --  Flag )
   (  --  N )
```

Figure 6.5: Reject a BEGIN-UNTIL loop that does not always produce a top flag value

Anyway, the literal `0` is used in the ELSE branch of `start-game` to indicate that the loop should continue in case the input could not be parsed as a number. We circumvent

```
: give-advice
    cr ." The requested number is "
    0 < if ." larger. "
        else ." smaller. " then cr ;

' give-advice ( assert xt:[ n -- ] ) explain-wrong-guess !
: feedback
    secret-number @ - dup
    0= dup
    if
        success
    else
        swap

        explain-wrong-guess @ ( assert xt:[ n -- ] ) execute

        ." Try again " cr
    then ;
```

Listing 6.6: Using indirection for factoring out functionality

the problem by using the cast `( Cast n -- flag )`, see Line 32 in Listing 6.1. As a matter of fact, it is only that cast annotation that is actually needed to make the checker succeed, all other types are inferred, including the type of `secret-number`.

```
create-random-nr ( -- N )
explain-wrong-guess ( -- *xt:[ N -- ] )
feedback ( N -- Flag )
give-advice ( N -- )
init-secret-number ( -- )
init-seed ( -- )
read-guess ( -- N Flag )
secret-number ( -- *N )
start-game ( -- )
success ( x1 x2 -- x2 )
wrong-input ( x -- N )
```

Figure 6.7: Final inferred effects of running checker4

The one area where annotations are always needed is the use of higher-order programming. As a possible real-world enhancement or refactoring, we factor part of `feedback` out into a reference to a function such that it is configurable how much to advise the player. That reference `explain-wrong-guess` thus references a word with the effect

`( n -- )`. Those changes are visible in Listing 6.6. In addition, we could get rid of the cast by replacing `0` with an explicit `1 0=`. Alternatively, we specify `n` being a subtype of `flag` in `checker4`. We can gain trust by disabling `allowCasts` and thus ruling out casts in our codebase. As a matter of fact, the *conservative* checker `checker4` now runs successfully.
Consequently, we have now achieved static type consistency since the rigorous configuration of checker `checker4` does not permit statically unsound features. Optionally, we could now add the inferred stack effects of Figure 6.7 as stack effect comments in order to document the intent of the defined words.

Above it has been shown how those checkers can guide the evolution of an existing program towards static type consistency. Obviously, the checkers can provide their benefits even easier when they are integrated into the development of a new program from scratch: otherwise necessary, possibly large modifications to later account for static type consistency are not needed when the checker is run routinely. In addition, providing stack effect comments for colon definitions as a rule is highly recommendable since the cause of error can then be determined more easily from the checker's output, in general. As experience has shown, assertions are useful in this regard, too – they allow for step-wise checking the intended stack image with the checker's computed stack image as explained in Section 5.7. Thus, assertions are a helpful debugging tool useful to narrow down the exact location of the error when the checker reports a mismatch between a colon definition's inferred effect and its specificied stack effect comment.

6.2 Comparing related work

6.2.1 Comparison of functionality

Limitations. The starting point of integrating Forth words has been the Forth CORE wordset. However, the use of some words of that wordset is not supported in the presented implementation. The following words are not supported as they excel the intended goals listed in Section 1.4: `2@`, `2!`, `state`, `recurse`, `r@`, `r>`, `does>`, `variable`, `constant`. In the case of the latter words, the more general `create` is supported for working with references while `does>`, `variable` and `constant` could be partly expressed in terms of `create`. In contrast, [Kna93] features support for those words in its design of the `FLINT` Forth type checker and checks recursive calls of `recurse` by comparing the stack image at the point of that word's occurrence with the stack effect specification. Similarly, support of the return stack (or other stacks as a floating point stack) for words like `r@` could be added without great effort. The words `2@` and `2!` are used for accessing references of a double-cell sized type which could be supported with considerable modifications of the stack effect calculus implementation. Furthermore, words as `unloop`, `quit`, `leave`, `exit`, `evaluate`, `environment?`, `abort`, `abort-quote` and `find` cannot be used with the implemented checkers. This is not surprising as they have

effects which are typically hardly amenable to static analysis. Supporting `abort` and `abort-quote` would have required modifying the stack effect calculus to a greater extent as those words work on an arbitrarily large stack image.

Stack Effect Calculus. The checker implementation uses the commonly used stack effect calculus as presentend in Section 5.3 as its theoretical basis. In contrast to previous works, that approach has been expanded to deal with multiple stack effects, input stream arguments, reference types, object-oriented programming and compile-time programming in such a way that accounts for static type consistency. In addition, casts and assertions as already proposed in [Kna93] have been successfully integrated into the calculus. Moreover, an annotation-based integration of higher-order programming has been outlined where the type of the function stack value needs to be specified by an assertion before execution. That requirement is partly due to the workings of the chosen stack calculus and partly due to the nature of the difficulty of compile-time type checking of higher-order programming. Doing without type annotations in *some* cases of higher-order programming would have been feasible if a Hindley-Milner variant, adapted to stack operations, had been used as a basis for type inference. In that case, however, stack operations involving higher-order programming would need to be expressed as at least Rank-2 types insofar as all stack operations are polymorphic in the part of the stack they leave untouched. While type inference for Rank-2 types is known to be decidable (see [Wel93]) it would be difficult to integrate it into the existing stack effect calculus. Having said that, there is the programming language `Cat` which promises simple type inference for stack-based languages (see [Dig08]). It makes use of a Hindley-Milner variant and claims to cope with Rank-2 type inference by introducing a rewriting rule in addition to core Hindley-Milner. However, since [Wel93] it is known that the type inference problem for types of rank greater or equal to 3 is undecidable anyway. Unfortunately, the necessity of Rank-3 types can easily arise in higher-order programming scenarios; consequently, it is not possible to provide full type inference for higher-order programming without type annotations in the general case. As a result, current research is rather done on how to implement higher-order type inference with as few annotations as possible (see [PJVWS07]).

Control Structures

IFELSE. The predominant approach for compile-time typing an IFELSE or IF expression in the literature is about enforcing or computing a common type of both branches from the beginning; likewise, this behavior holds if the configuration option `allowMultipleEffects` is set to `False`. This has been a conscious design decision – as the implemented stack effect calculus can deal with multiple stack effects this allows for a more gradual typing process given an existing Forth program. Consequently, it is the user's choice to enforce a static typing approach by disallowing multiple effects by configuration. In that case, it is attempted to derive a common type of both branches with the approach of Section 5.6. [Pö03]

deals better with that problem by making use of a typing rule for the computation of a greatest lower bound of stack effects.

Loops. The body of loop expressions is checked to leave the stack untouched as there is no compile-time knowledge as to how often a loop is executed. This is a stringent requirement mentioned in all other works on static Forth analysis and is detailled in [SK93].

6.2.2 Optional, pluggable types

In comparison with similar projects for adding optional typing to an existing language, the implemented Forth type checker rather resembles the TypePlug system's [HDN09] approach: A checker is created by overriding options in some sort of configuration while the stack effect composition algorithm stays unchangeable as opposed to specifying type system rules as in [MME+10].

The implementation of the stack effect calculus providing type inference and type checking adheres to the restrictions of an optional, pluggable type system proposed in [Bra04]: Firstly, the type system trivially has no effect on the runtime-semantics of the programming language and secondly, it does not mandate any type annotations in the syntax – stack effect comments are optional in colon definitions. However, there is a language feature where type annotations are needed, namely higher-order programming as explained in Section 6.2.1.

CHAPTER 7

Conclusion

In chapters 2 and 3 an overview of the implications of type systems enforcing different degrees of static type consistency has been given. In doing so, the qualities and differences of representatives of soft typing, gradual typing and optional typing were analyzed in order to motivate the use of optional, pluggable typing for the Forth programming language.

Chapter 4 has elaborated on the characteristics of Forth concerning its stack-based computational model. Having those in mind, Chapter 5 provides the design of the optional, pluggable typing prototype for Forth together with an implementation in Haskell.

Related work on static Forth program analysis has been discussed in order to find a theoretical basis for type checking and type inference in the form of a stack effect composition algorithm. Thus, a simple stack effect calculus has been introduced and implemented. The implementation has then been enhanced to deal with input stream arguments, multiple stack effects, compile-time programming, subtyping, reference types, respectively. In addition, a simple object-oriented programming extension as well as higher-order programming were integrated. Generally, the problem of type inference has turned out to be easier to handle in Forth than in other languages. Type annotations in the form of stack effect comments provide optional, checked documentation of a colon definitions effect. It is only in the field of higher-order programming where annotations have proved necessary to assist the stack effect composition algorithm.

In Chapter 6 the limitations of the resulting type checker have been listed and it has been practically evaluated. Certain words of the targeted Forth CORE wordset could not be integrated, mostly due to their dynamic semantics. Besides, support for the return stack, double-cell types and the runtime semantics of created references with `does>` would be considered to be useful in future work. Moreover, various improvements to the stack effect language in use would be beneficial, e.g. the feature to give names to the values standing for the types in a stack effect.

In the context of optional typing approaches, the checker has been considered to be similar to *TypePlug* [HDN09] in so far as the type checker features a fixed type checking algorithm where types are "plugged in" by configuration options.

Finally, it has been shown how the system can be configured to create checkers of varying type safety which were applied to a small, functionally correct Forth program as a demonstration of their practical use. The tested checkers range from a configuration only checking for stack overflow to a configuration allowing multiple effects, casts and assertions. At last, a strict configuration enforcing *conservative* type consistency has been used in that process. In that way, it has been demonstrated how the checkers identify typical violations of static type consistency within the scope of functionally correct Forth code. In addition, that example successfully illustrates how the user is guided from a checker's output to change the input program to account for more static type consistency, thus gradually increasing the degree of compile-time type consistency as part of the Forth developing process.

Bibliography

[ACF+13] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

[ans94] ANS Forth-1994. Downloaded from http://www.forth.com/downloads/dpans94.pdf on 07/11/2015., 1994.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 215–230, New York, NY, USA, 1993. ACM.

[Bo81] Barry W. Boehm and others. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[Bra04] Gilad Bracha. Pluggable type systems. In *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[BS12] Ambrose Bonnaire-Sergeant. *A Practical Optional Type System for Clojure*. PhD thesis, The University of Western Australia, 2012.

[Car96] Luca Cardelli. Type Systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. *ACM SIGPLAN Notices*, 26(6):278–292, 1991.

[CHH09] Patrick Camphuijsen, Jurriaan Hage, and Stefan Holdermans. Soft typing PHP. Technical report, Technical report, Utrecht University, 2009.

[CK14] Paolo Capriotti and Ambrus Kaposi. Free applicative functors. *arXiv preprint arXiv:1403.0749*, 2014.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[DDES11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, and Todd W. Schiller. *Building and Using Pluggable Type-Checkers*. 2011.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-Method Reflection. In Jean Bézivin and Bertrand Meyer, editor, *TOOLS Europe 2007*, volume 6/9, pages 231–251, Zürich, Switzerland, 2007. JOT.

[Dig08] Christopher Diggins. Simple Type Inference for Higher-Order Stack-Oriented Languages. Downloaded from http://bit.ly/1gudwu2 on 07/11/2015., 2008.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *ACM SIGPLAN Notices*, volume 30, pages 169–184. ACM, 1995.

[FAFH09] Michael Furr, Jong-hoon David An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866. ACM, 2009.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, June 1990.

[Gra06] Martin Grabmüller. *Algorithm W Step by Step. Downloaded from http://bit.ly/1Mnme5G on 07/11/2015.* 2006.

[HDN09] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical pluggable types for a dynamic language. *Comput. Lang. Syst. Struct.*, 35:48–62, 2009.

[HP99] Haruo Hosoya and Benjamin C. Pierce. How Good is Local Type Inference? pages 252–265, 1999.

[HTF10] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, June 2010.

[II09] Lintaro Ina and Atsushi Igarashi. Towards Gradual Typing for Generics. In *Proceedings for the 1st Workshop on Script to Program Evolution*, STOP '09, pages 17–29, New York, NY, USA, 2009. ACM.

[II11] Lintaro Ina and Atsushi Igarashi. Gradual Typing for Generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM.

[jav10] JavaCOP Tutorial. http://javacop.sourceforge.net/docs/tutorial.html, 2010.

[JG97] Mehdi Jazayeri and Carlo GHEZZI. *Programming language concepts*. New York: Wiley, 1997.

[Jim96] Trevor Jim. What Are Principal Typings and What Are They Good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 42–53, New York, NY, USA, 1996. ACM.

[KJS10] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

[Kna93] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, The University of Teesside, 1993.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-unification Problem. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 468–476, New York, NY, USA, 1990. ACM.

[LG11] Jukka Lehtosalo and David J. Greaves. Language with a Pluggable Type System and Optional Runtime Monitoring of Type Errors. In *Proceedings of International Workshop on Scripts to Programs (STOP)*, 2011.

[MD04] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of functional programming*, 1(03):245–285, 1991.

[Mit95] John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 176–185. ACM, 1995.

[MME+10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4, 2010.

[MMI14] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An Optional Type System for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–10. ACM, 2014.

[Nys03] Sven-Olof Nyström. A soft-typing system for Erlang. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71. ACM, 2003.

[PACJ+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[Pie91] Benjamin C. Pierce. Programming with Intersection Types and Bounded Polymorphism. Technical report, 1991.

[PJVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical Type Inference for Arbitrary-rank Types. *J. Funct. Program.*, 17(1):1–82, January 2007.

[Poi91] Jannus Poial. Multiple Stack-effects of Forth Programs. In *1991 FORML Conference Proceedings, euroFORML*, volume 91, pages 11–13, 1991.

[Pö90] Jaanus Pöial. Algebraic Specifications of Stack Effects for Forth Programs. In *EuroFORML'90 Conference Proceedings*, 1990.

[Pö02] Jaanus Pöial. Stack effect calculus with typed wildcards, polymorphism and inheritance. In *Proc. 18-th EuroForth Conference*, page 38, 2002.

[Pö03] J. Pöial. Program analysis for stack based languages. In *EuroFORTH conference on the FORTH programming language and FORTH processors, Herefordshire, UK (October 17-19, 2003)*, 2003.

[Pö06] Jaanus Pöial. Typing Tools for Typeless Stack Languages. In *22nd EuroForth Conference*, page 40, 2006.

[Pö08] Jaanus Pöial. Java Framework for Static Analysis of Forth Programs. In *Proc. 24-th EuroForth Conference*, 2008.

[RCH12] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 481–494, New York, NY, USA, 2012. ACM.

[RP96] Bradford J. Rodriquez and W. F. S. Poehlman. A Survey of Object-oriented Forths. *SIGPLAN Not.*, 31(4):39–42, April 1996.

[RSF+14] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. 2014.

[SDD+04] J. Stecklein, Jim Dabney, B. Dick, Bill Haskins, Randy Lovell, and Gregory Moroney. Error cost escalation through the project life cycle. *National Aeronautics and Space Administration*, 2004.

[SG12] Jeremy G. Siek and Ronald Garcia. Interpretations of the Gradually-typed Lambda Calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pages 68–80, New York, NY, USA, 2012. ACM.

[SGT09] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the Design Space of Higher-Order Casts. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 17–31, Berlin, Heidelberg, 2009. Springer-Verlag.

[SK93] Bill Stoddart and Peter J. Knaggs. Type Inference in Stack Based Languages. *Formal Aspects of Computing*, 5:289–298, 1993.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[ST07] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*, number 4609 in Lecture Notes in Computer Science, pages 2–27. Springer Berlin Heidelberg, January 2007.

[SV08] Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM.

[SVB13] Jeremy G. Siek, Michael M. Vitousek, and Shashank Bharadwaj. *Gradual Typing for Mutable Objects*. 2013.

[Tha90] Satish Thatte. Quasi-static Typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 367–381, New York, NY, USA, 1990. ACM.

[THF08] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

[Unt12] Martin Unterholzner. Refactoring Support for Smalltalk Using Static Type Inference. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '12, pages 1:1–1:18, New York, NY, USA, 2012. ACM.

[WC97] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):87–152, 1997.

[Wel93] Joe Wells. A Direct Algorithm for Type Inference in the Rank 2 Fragment of the Second-Order Lambda-Calculus. Technical report, Boston University, Boston, MA, USA, 1993.

[Wes02] J. Christopher Westland. The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1):1–9, 2002.