# Sage: Hybrid Checking for Flexible Speciﬁcations

Abstract

Software systems typically contain large APIs that are in-formally speciﬁed and hence easily misused. This paperpresents the Sage programming language, which is designedto enforce precise interface speciﬁcations in a ﬂexible man-ner. The Sage type system uses a synthesis of the type Dynamic, ﬁrst-class types, and arbitrary reﬁnement types. Since type checking for this expressive language is not stat-ically decidable, Sage uses hybrid type checking, which ex-tends static type checking with dynamic contract checking,automatic theorem proving, and a database of refuted sub-type judgments.

1. Introduction

Constructing a large, reliable software system is extremely challenging, due to the diﬃculty of understanding the sys-tem in its entirety. A necessary strategy for controlling this conceptual complexity is to divide the system into modules that communicate via clearly speciﬁed interfaces. The precision of these interface speciﬁcations may natu-rally and appropriately evolve during the course of software development. To illustrate this potential variation, consider the following speciﬁcations for the argument to a function

invertMatrix:

1. The argument can be any (dynamically-typed) value.

2. The argument must be an array of arrays of numbers.

3. The argument must be a matrix, that is, a rectangular (non-ragged) array of arrays of numbers.

4. The argument must be a square matrix.

5. The argument must be a square matrix that satisﬁes the predicate isInvertible.

All of these speciﬁcations are valid constraints on the ar-gument to invertMatrix, although some are obviously more precise than others. Diﬀerent speciﬁcations may be appro-priate at diﬀerent stages of the development process. Simpler speciﬁcations facilitate rapid prototyping, whereas more pre-cise speciﬁcations provide more correctness guarantees and better documentation. Traditional statically-typed languages, such as Java, C#, and ML, primarily support the second of these speciﬁca-tions. Dynamically-typed languages such as Scheme primar-ily support the ﬁrst speciﬁcation. Contracts [32, 13, 26,21, 24, 28, 37, 25, 12, 8] provide a means to document and enforce all of these speciﬁcations, but violations are only detected dynamically, resulting in incomplete and late (possibly post-deployment) detection of defects. This paper presents the Sage programming language and type system, which is designed to support and enforce a wide range of speciﬁcation methodologies. Sage veriﬁes correctness prop-erties and detects defects via static checking wherever pos-sible. However, Sage can also enforce speciﬁcations dynam-ically, when necessary.

On a technical level, the Sage type system can be viewed as a synthesis of three concepts: the type Dynamic; arbitrary reﬁnement types; and ﬁrst-class types. These features add expressive power in three orthogonal directions, yet they all cooperate neatly within Sage’s hybrid static/dynamic checking framework.

Type Dynamic. The type Dynamic [23, 1, 39] enables Sage to support dynamically-typed programming. Dynamic is a supertype of all types; any value can be upcast to type Dynamic, and a value of declared type Dynamic can be implicitly downcast (via a run-time check) to a more precise type. Such downcasts are implicitly inserted when necessary, such as when the operation add1 (which expects an Int) is applied to a variable of type Dynamic. Thus, declaring variables to have type Dynamic (which is the default if type annotations are omitted) leads to a dynamically-typed, Scheme-like style of programming.

These dynamically-typed programs can later be anno-tated with traditional type speciﬁcations like Int and Bool. One nice aspect of our system is that the programmer need not fully annotate the program with types in order to reap some beneﬁt. Types enable Sage to check more properties statically, but it is still able to fall back to dynamic checking whenever the type Dynamic is encountered.

Reﬁnement Types. For increased precision, Sage also supports reﬁnement types. For example, the following code snippet deﬁnes the type of integers in the range from lo (inclusive) to hi (exclusive):

{ x:Int | lo <= x && x < hi }

Sage extends prior work on decidable reﬁnement types [44, 43, 18, 30, 35] to support arbitrary executable reﬁnement predicates — any boolean expression can be used as a reﬁnement predicate.

First-Class Types. Finally, Sage elevates types to be ﬁrst-class values, in the tradition of Pure Type Systems [5].

Thus, types can be returned from functions, which permits function abstractions to abstract over types as well as terms. For example, the following function Range takes two integers and returns the type of integers within that range:

let Range (lo:Int) (hi:Int) : \* = { x:Int | lo <= x && x < hi };

Here, \* is the type of types and indicates that Range returns a type. Similarly, we can pass types to functions, as in the following polymorphic identity function:

let id (T:\*) (x:T) : T = x;

The traditional limitation of both ﬁrst-class types and unrestricted reﬁnement types is that they are not stati-cally decidable. Sage circumvents this diﬃculty by replacing static type checking with hybrid type checking [14]. Sage checks correctness properties and detects defects statically, whenever possible. However, it resorts to dynamic checking for particularly complicated speciﬁcations. The overall re-sult is that precise speciﬁcations can be enforced, with most errors detected at compile time, and violations of some com-plicated speciﬁcations detected at run time.

# Phantom Types

Abstract. Phantom types are data types with type constraints asso-ciated with diﬀerent cases. Examples of phantom types include typed type representations and typed higher-order abstract syntax trees. These

types can be used to support typed generic functions, dynamic typing, and staged compilation in higher-order, statically typed languages such as Haskell or Standard ML. In our system, type constraints can be equa-tions between type constructors as well as type functions of higher-order kinds. We prove type soundness and decidability for a Haskell-like lan-guage extended by phantom types.

1 Introduction

Generic functions, dynamic typing, and metacircular interpretation are powerful features found in dynamically typed programming languages but have proved diﬃcult to incorporate into statically typed languages such as Haskell or Standard ML. Past approaches have included adding a ﬁrst-class Dynamic type and typecase expressions [1, 2, 19], deﬁning generic functions by translation from “polytypic” languages to existing languages [15, 13], and implementing staged computation with run-time type checking [8, 23] or compile-time computation [22]. Recently, Cheney and Hinze [6] and Baars and Swierstra [4] found that many of these features can already be implemented via an encoding into Haskell based on equality types comprising “proofs” of type equality.

However, this encoding has several drawbacks:

– It imposes a high annotation burden on programmers, limiting its usability

– it incurs unnecessary run-time overhead in the form of calls to the identity function;

– it requires a diﬀerent equality type deﬁnition for each kind; and

– it is limited by the constraint-solving abilities of the underlying type-checked

The last limitation is particularly vexing because it places many interesting potential applications just out of reach, and imposes unnecessary run-time overhead in others. For example, some type-indexed types (e.g. generic generalized tries [12] and Typerec [10, 7]) can almost, but not quite, be implemented naturally.

The obstacle is that the type checker does not know certain valid properties of types that are not important for normal type checking. For example, because type equality in Haskell is syntactic, τ1 = τ 0

1 follows from τ1×τ2 = τ 0 1×τ 0

2, but in

Haskell, there is no useful way to “prove” this implication using equality types.

We will discuss this point in more detail in Sections 2.3 and 4. In this paper we present a programming construct for deﬁning so-called phan-tom types. In our formalism, data type cases may be annotated by with clauses of the form with {τ1 = τ 0 1; . . . ; τn = τ 0n}. The equations listed in a with clause must hold whenever the corresponding constructor is applied and may be used in type checking case bodies for the constructor. In combination with unrestricted existential quantiﬁers, data type deﬁnitions using type equations can be used to implement both introduction and elimination forms for many advanced ap-plications of phantom types, including typed type representations and typed higher-order abstract syntax. The term phantom type was originally coined by Leijen and Meijer [17] to denote parameterized types that do not use their type argument(s). Theyuse phantom types for type-safe embeddings of domain speciﬁc languages into Haskell. Their approach is, however, strictly more limited: while they can enforce type constraints when constructing values of a phantom type, they cannot make use of these constraints when decomposing a value. Our proposal exactly ﬁlls this gap. In the rest of this paper, we describe our proposal in more detail, discuss applications, present a type system for a Haskell-like language that has phantom types, and prove some of its properties. Finally, we discuss related and future work and conclude.

# Fun with type functions

Abstract

Tony Hoare has always been a leader in writing down and proving prop-

erties of programs. To prove properties of programs automatically, the

most widely used technology today is by far the ubiquitous type checker.

Alas, static type systems inevitably exclude some good programs and

allow some bad ones. Thus motivated, we describe some fun we have

been having with Haskell, by making the type system more expressive

without losing the beneﬁts of automatic proof and compact expression.

Speciﬁcally, we oﬀer a programmer’s tour of so-called type families, a re-

cent extension to Haskell that allows functions on types to be expressed

as straightforwardly as functions on values. This facility makes it easier

for programmers to eﬀectively extend the compiler by writing functional

programs that execute during type-checking.

This paper gives a programmer’s tour of type families as they are

supported in GHC.

This paper appears in the proceedings of Tony Hoare’s 75th birth-

day Festschrift, to be published by Springer in the “History of Com-

puting” series in 2010. This online version of the paper includes mi-

nor improvements and, more substantially, an Appendix that does not

appear in the published version. Source code for all the examples is

available at http://research.microsoft.com/~simonpj/papers/assoc-

types/fun-with-type-funs/fun-with-type-funs.zip

1 Introduction

The type of a function speciﬁes (partially) what it does. Although weak

as a speciﬁcation language, static types have compensating virtues: they

are

• lightweight, so programmers use them;

• machine-checked with minimal programmer assistance;

• ubiquitous, so programmers cannot avoid them.

As a result, static type checking is by far the most widely used veriﬁcation

technology today.

Every type system excludes some “good” programs, and permits some

“bad” ones. For example, a language that lacks polymorphism will reject

this “good” program:

f :: [Int] -> [Bool] -> Int

f is bs = length is + length bs

Why? Because the length function cannot apply to both a list of Ints and

a list of Bools. The solution is to use a more sophisticated type system

in which we can give length a polymorphic type.

Conversely, most languages will accept the expression

speed + distance

where speed is a variable representing speed, and distance represents

distance, even though adding a speed to a distance is as much nonsense

as adding a character to a boolean.

The type-system designer wants to accommodate more good programs

and exclude more bad ones, without going overboard and losing the virtues

mentioned above. In this paper we describe type families, an experimen-

tal addition to Haskell with precisely this goal. We start by using type

families to accommodate more good programs, then turn in Section 5 to

excluding more bad programs. We focus on the programmer, and our

style is informal and tutorial. The technical background can be found

elsewhere [5–7, 42]. The complete code described in the paper is available

. That directory also contains the online version of the paper with addi-

tional appendices, brieﬂy describing the syntax of type functions and the

rules and pitfalls of their use. Appendix C gives an alternative derivation

of typed sprintf using higher-order type-level functions.

# How to Declare an Imperative

# A theory of qualified types

# Linear types can change the world!

The linear logic of J.-Y. Girard suggests a new type system for functional languages, one which supports operations that \change the world". Values be-longing to a linear type must be used exactly once: like the world, they cannot be duplicated or destroyed. Such values require no reference counting or garbage col-lection, and safely admit destructive array update. Linear types extend Schmidt's notion of single threading; provide an alternative to Hudak and Bloss' update analysis; and offer a practical complement to Lafont and Holmstr om's elegant linear languages.

# Typed Memory Management in a Calculus of Capabilities∗

Abstract

An increasing number of systems rely on programming lan-guage technology to ensure safety and security of low-level code. Unfortunately, these systems typically rely on a com-plex, trusted garbage collector. Region-based type systems provide an alternative to garbage collection by making mem-ory management explicit but veriably safe. However, it has not been clear how to use regions in low-level, type-safe code. We present a compiler intermediate language, called the Capability Calculus, that supports region-based memory management, enjoys a provably safe type system, and is straightforward to compile to a typed assembly language. Source languages may be compiled to our language using known region inference algorithms. Furthermore, region life-times need not be lexically scoped in our language, yet the language may be checked for safety without complex anal-yses. Finally, our soundness proof is relatively simple, em-ploying only standard techniques.

The central novelty is the use of static capabilities to specify the permissibility of various operations, such as memory access and deallocation. In order to ensure ca-pabilities are relinquished properly, the type system tracks aliasing information using a form of bounded quantication.

1 Motivation and Background

A current trend in systems software is to allow untrusted ex-tensions to be installed in protected services, relying upon language technology to protect the integrity of the service instead of hardware-based protection mechanisms [19, 39, 2,25, 24, 17, 14]. For example, the SPIN project [2] relies upon the Modula-3 type system to protect an operating system kernel from erroneous extensions. Similarly, web browsers rely upon the Java Virtual Machine byte-code verifer [19] to protect users from malicious applets. In both situations, the goal is to eliminate expensive communications or boundary crossings by allowing extensions to directly access the re-sources they require.

Recently, Necula and Lee [26, 25] have proposed Proof-Carrying Code (PCC) and Morrisett et al. [24] have sug-gested Typed Assembly Language (TAL) as language tech-nologies that provide the security advantages of high-level languages, but without the overheads of interpretation or just-in-time compilation. In both systems, low-level ma-chine code can be heavily optimized, by hand or by com-piler, and yet be automatically verifed through proof- or type-checking. However, in all of these systems (SPIN, JVM, TAL, and Touchstone [27], a compiler that generates PCC), there is one aspect over which programmers and optimizing compil-ers have little or no control: memory management. In par-ticular, their soundness depends on memory being reclaimed by a trusted garbage collector. Hence, applets or kernel extensions may not perform their own optimized memory management. Furthermore, as garbage collectors tend to be large, complicated pieces of unverified software, the degree of trust in language-based protection mechanisms is dimin-shed. The goal of this work is to provide a high degree of con-trol over memory management for programmers and com-pilers, but as in the PCC and TAL frameworks, make veri-fication of the safety of programs a straightforward task.

# Lightweight static capabilities

Abstract

We describe a modular programming style that harnesses modern type systems to verify safety conditions in practical systems. This style has three ingredients:

(i) A compact kernel of trust that is specific to the problem domain.

(ii) Unique names (capabilities) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.

(iii) Static (type) proxies for dynamic values.

We illustrate our approach using examples from the dependent-type literature, but our programs are written in Haskell and OCaml today, so our techniques are compatible with imperative code, native mutable arrays, and general recursion. The three ingredients of this programming style call for (1) an expressive core language, (2) higher-rank polymorphism, and (3) phantom types.

# Monadic Regions

Un sacco di roba, soprattutto un gran riassunto delle Regions originali

# Lightweight Monadic Regions

Abstract

We present Haskell libraries that statically ensure the safe use of resources such as ﬁle handles. We statically prevent accessing an already closed handle or forgetting to close it. The libraries can be trivially extended to other resources such as database connections and graphic contexts. Because ﬁle handles and similar resources are scarce, we want to not just assure their safe use but further deallocate them soon after they are no longer needed. Relying on Fluet and Morrisett’s [4] calculus of nested regions, we contribute a novel, improved, and extended implementation of the calculus in Haskell, with ﬁle handles as resources. Our library supports region polymorphism and implicit region subtyping, along with higher-order functions, mutable state, recur-sion, and run-time exceptions. A program may allocate arbitrarily many resources and dispose of them in any order, not necessarily LIFO. Region annotations are part of an expression’s inferred type. Our new Haskell encoding of monadic regions as monad trans-formers needs no witness terms. It assures timely deallocation even when resources have markedly different lifetimes and the identity of the longest-living resource is determined only dynamically. For contrast, we also implement a Haskell library for manual resource management, where deallocation is explicit and safety is assured by a form of linear types. We implement the linear typing in Haskell with the help of phantom types and a parameterized monad to statically track the type-state of resources.

# Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages

This paper argues that we should seek the golden middle way between dynamically and statically typed languages.

# Strongly Typed Memory Areas

Modern functional languages offer several attractive features to support development of reliable and secure software. However, in our efforts to use Haskell for systems programming tasks—including device driver and operating system construction—we have also encountered some signiﬁcant gaps in functionality. As a result, we have been forced, either to code some non-trivial components in more traditional but unsafe languages like C or assembler, or else to adopt aspects of the foreign function interface that compromise on strong typing and type safety. In this paper, we describe how we have ﬁlled one of these gaps by extending a Haskell-like language with facilities for working directly with low-level, memory-based data structures. Using this extension, we are able to program a wide range of examples, in-cluding hardware interfaces, kernel data structures, and operating system APIs. Our design allows us to address concerns about repre-sentation, alignment, and placement (in virtual or physical address spaces) that are critical in some systems applications, but clearly beyond the scope of most existing functional languages. Our approach leverages type system features that are well-known and widely supported in existing Haskell implementations, including kinds, multiple parameter type classes, functional depen-dencies, and improvement. One interesting feature is the use of a syntactic abbreviation that makes it easy to deﬁne and work with functions at the type level.

# Position: Lightweight static resources; Sexy types for embedded and systems programming

Abstract

It is an established trend to develop low-level code—embedded software, device drivers, and operating systems—using high-level languages, especially functional languages with advanced facilities to abstract and generate code. To be reliable and secure, low-level code must correctly manage space, time, and other resources, so special type systems and veriﬁcation tools arose to regulate resource access statically. However, a general-purpose functional language practical today can provide the same static assurances, also without run-time overhead. We substantiate this claim and promote the trend with two security kernels in the domain of device drivers:

1. one built around raw pointers, to track and arbitrate the size, alignment, write permission, and other properties of memory areas across indexing and casting;

2. the other built around a device register, to enforce protocol and timing require-ments while reading from the register.

Our style is convenient in Haskell thanks to custom kinds and predicates (as type classes); type-level numbers, functions, and records (using functional dependencies); and mixed type- and term-level programming (enabling partial type signatures).

**Adoption and Focus: Practical Linear Types for Imperative Programming**

ABSTRACT

A type system with linearity is useful for checking soft-ware protocols and resource management at compile time. Linearity provides powerful reasoning about state changes, but at the price of restrictions on aliasing. The hard division between linear and nonlinear types forces the programmer to make a trade-oﬀ between checking a protocol on an object and aliasing the object. Most onerous is the restriction that any type with a linear component must itself be linear. Because of this, check-ing a protocol on an object imposes aliasing restrictions on any data structure that directly or indirectly points to the object. We propose a new type system that re-duces these restrictions with the adoption and focus con-structs. Adoption safely allows a programmer to alias objects on which she is checking protocols, and focus allows the reverse. A programmer can alias data struc- tures that point to linear objects and use focus for safe access to those objects. We discuss how we implemented these ideas in the Vault programming language.

# Typed Memory Management in a Calculus of Capabilities∗

An increasing number of systems rely on programming lan-guage technology to ensure safety and security of low-level code. Unfortunately, these systems typically rely on a com-plex, trusted garbage collector. Region-based type systems provide an alternative to garbage collection by making mem-ory management explicit but verifiably safe. However, it has not been clear how to use regions in low-level, type-safe code.

We present a compiler intermediate language, called the Capability Calculus, that supports region-based memory management, enjoys a provably safe type system, and is straightforward to compile to a typed assembly language. Source languages may be compiled to our language using known region inference algorithms. Furthermore, region life-times need not be lexically scoped in our language, yet the language may be checked for safety without complex anal-yses. Finally, our soundness proof is relatively simple, em-

ploying only standard techniques. The central novelty is the use of static capabilities to specify the permissibility of various operations, such as memory access and deallocation. In order to ensure ca- pabilities are relinquished properly, the type system tracks aliasing information using a form of bounded quantification.