Pontifícia Universidade Católica
do Rio de Janeiro

Hugo Musso Gualandi

Typing Dynamic Languages – a Review

DISSERTAÇÃO DE MESTRADO

Dissertation presented to the Programa de Pós-Graduação em Informática at PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
September 2015

Hugo Musso Gualandi

Typing Dynamic Languages – a Review

Dissertation presented to the Programa de Pós-Graduação em Informática at PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the examining commission signed below.

Rio de Janeiro, September 8th, 2015

Hugo Musso Gualandi

Graduated in 2011 with a Bachelor's Degree in Molecular Sciences by the University of São Paulo (USP).

Ficha Catalográfica

Gualandi, Hugo Musso

Typing dynamic languages : a review / Hugo Musso Gualandi ; advisor: Roberto Ierusalimschy. – 2015.
90 f. : il. color. ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.
Inclui bibliografia

1. Informática – Teses. 2. Linguagens dinâmicas. 3. Sistemas de tipos. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgements

This work was funded by CNPq grant number 146516/2012-5.

Abstract

Gualandi, Hugo Musso ; Ierusalimschy, Roberto . **Typing Dynamic Languages – a Review**. Rio de Janeiro, 2015. 90p. MSc. Dissertation – Departamento de Informática , PUC-Rio.

Programming languages have traditionally been classified as either statically typed or dynamically typed, the latter often being known as scripting languages. Dynamically typed languages are very popular for writing smaller programs, a setting where ease of use and flexibility of the language are highly valued. However, with time, small scripts tend to evolve into large systems and the flexibility of the dynamic language may become a source of program defects. For these larger systems, static typing, which offers compile-time error detection, improved documentation and optimization opportunities, becomes more attractive. Since rewriting the whole system in a statically typed language is not ideal from a software engineering point of view, investigating ways of adding static types to existing dynamically typed programs has been a thriving research area. In this work, we present a historical overview of this research. We focus on general approaches that apply to multiple programming languages, such as the Type Hints of Common LISP, the Soft Typing of Fagan et al and the Gradual Typing of Siek et al, contrasting these different solutions from a modern perspective.

Keywords
Dynamic languages; Type systems.

Resumo

Gualandi, Hugo Musso ; Ierusalimschy, Roberto . **Tipando Linguagens Dinâmicas – uma Revisão**. Rio de Janeiro, 2015. 90p. Dissertação de Mestrado – Departamento de Informática , PUC-Rio.

Linguagens de programação tem tradicionalmente sido classificadas como estaticamente tipadas ou dinamicamente tipadas, estas últimas também sendo conhecidas como linguagens de scripting. Linguagens com tipagem dinâmica são bastante populares para a escrita de programas menores, um cenário onde a facilidade de uso e flexibilidade da linguagem são altamente valorizados. No entanto, com o passar do tempo, pequenos scripts podem se tornar grandes sistemas e a flexibilidade da linguagem pode passar a ser uma fonte de defeitos no programa. Para estes sistemas maiores, a tipagem estática, que oferece detecção de erros em tempo de compilação, melhor documentação e oportunidades de otimização, passa a ser mais atrativa. Como reescrever todo o sistema em uma linguagem estática não é ideal do ponto de um vista da engenharia de software, encontrar formas de adicionar tipos estáticos em programas dinamicamente tipados já existentes tem sido uma área de pesquisa bem rica. Nesse trabalho, nós apresentamos uma perspectiva histórica dessa pesquisa. Nos focamos em abordagens que não são específicas para uma única linguagem de programação, como as Type Hints de Common LISP, o Soft Typing de Fagan et al e o Gradual Typing de Siek et al, contrastando essas diferentes soluções a partir de uma perspectiva moderna.

Palavras-chave

Linguagens dinâmicas; Sistemas de tipos

Contents

All is fair in love and war, even trying to add a static type system in a dynamically-typed programming language.

Lindahl \& Sagonas

1 Introduction

Historically, programming languages have been classified as either statically-typed or dynamically-typed . The simplicity and flexibility of dynamic languages makes them popular for writing prototypes and other small scripts. However, as programs grow larger, maintenance and debugging costs increase, which causes type systems and other forms of static program analysis to become more appealing. Unfortunately, converting an existing untyped program into a statically-typed one usually means rewriting the program in a completely different programming language, which is a costly undertaking.

This begs the question: Is it possible to make the transition from a dynamic program to a static one easier, therefore allowing the programmer to simultaneously benefit from the flexibility of dynamic languages and the static guarantees of typed ones?

There is a vast literature on this topic, ranging from developing static type systems that are expressive enough to reason about common dynamic idioms to ways of embedding dynamically-typed programs inside statically-typed ones. In this work we focus on the problem of adding a static type system to a dynamically typed language and we aim to provide an overview of some of the most important techniques used to achieve this goal.

In Section 1.1 and Section 1.2 we start by motivating the combination of static and dynamic typing. Why should we bother with combining them instead of just sticking to one or the other from the start? In Chapter 2 we present a quick review of type systems and programming language theory. In addition to clarifying the notation that we will be using throughout this text, we also formally describe what it means for a programming language to be statically-typed or dynamically-typed. Chapter 3 reviews the type system theory behind some type system features that do not affect the evaluation semantics of the underlying programming language. These techniques are used to make type systems more expressive and many of them are commonly used in type systems for dynamic languages. While we cannot give a comprehensive review of all the type systems for dynamic languages, we can describe some of the more common features among them.

After the theory chapters we move on to presenting some of the major branches of type systems for dynamic languages. Chapter 4 is about Soft Typing, a form of static analysis for dynamic languages that is based around automatic type inference. In Chapter 5 we present the optional type declarations of Common LISP, which was one of the first dynamic languages to allow optional type annotations. In Chapter 6 we discuss Gradual Typing, which is a way to create programs that are divided into typed and untyped regions in a way that prevents the untyped parts from violating the static guarantees given by the statically-typed parts. Finally, in Chapter 7, we cover Optional Typing, which is what happens when a type system has optional type annotations that do not affect the runtime semantics of programs.

1.1 The Advantages of Static Typing

Despite the great variety of existing type systems, even in the dynamic typing literature there is a lot of agreement over what are the main benefits of static typing. Some of the benefits that are mentioned most often are the following:

Early detection of programmer errors

Like any other form of static program analysis, type systems can detect program errors at compile time, without having to execute the program. This early detection can shorten the software development cycle, and allows programmers to be bolder while refactoring code. For example, if the interface of a subroutine is changed by adding an additional input argument, a type system can help the programmer by pointing out all the call sites that need to be updated to pass the additional parameter.

Another general characteristic of static analysis when it comes to error detection is that it does not depend on the program input. This is in contrast to runtime testing, which offers no guarantees about how the program will behave on the untested inputs.

Documentation and abstraction

Even in programming languages without a formal type system, it is common to use a type-based vocabulary to informally describe the contents of variables, subroutine inputs and outputs, module APIs and other interfaces. For example, in the Javascript documentation generator JSDoc [1] can parse type annotations inside comments and these type annotations can use a rich vocabulary including primitive types, union types and object types. This is despite the fact that the annotations are purely used for documentation and are completely ignored when the program is actually executed.

While these informal types are already useful, types can really shine when they are automatically checked by the computer, which can guarantee that the type declarations are always kept up to date. Information inside comments can easily become outdated after the program code next to goes through a rewrite.

External tooling

Types and type declarations are a form of metadata that can be used by a variety of tools other than the type checker. One of the most visible examples of this is how many text editors use type information to power autocompletion functionality and other forms of inline documentation.

Another interesting examples is the Hoogle [2] tool for the Haskell language, which can search though large APIs to find all functions that match a given type. This allows the programmer to find the function he wants based on the types of its inputs and outputs, even if he does not remember its name.

Efficiency

Type information can also allow the compiler to perform program optimizations or to allocate memory more efficiently. For example, compile-time type information can allow the compiled program to use untagged representations for data and avoid the branching operations that typically accompany tag tests in dynamic languages.

1.2 The Advantages of Dynamic Typing

The main advantage of dynamic languages is that they tend to be simpler and more flexible than static languages. Unfortunately, dynamic typing isn't formally defined like static typing is so most of the points that we list here as advantages of dynamic typing may be seen as disadvantages of a static type system.

Expressivity

As we will discuss in Section 2.5, any type system must be conservative and reject some well-behaved programs, which would execute without runtime errors if not for the type checker disallowing them from running at all. This is not just a theoretical limitation because many programming patterns that are possible to write in dynamic languages may require explicit support from the type system to be used in a static language.

For example, writing a type-generic data structure in a dynamic language is just a matter of writing code that does not directly inspect the elements stored in the data structure. On the other hand, in a static language its impossible to do the same unless the type system has been designed to support type polymorphism. Additionally, in Chapter 3 we also list more type system features that are meant to allow static languages to express programming patterns that come "for free" in dynamic languages.

Complexity and learning curve

One positive characteristic of dynamic languages is that the programmer only has to be aware of the language features that he is actually using in his program, which may not be the case in a statically-typed language this might not be the case. Firstly, if the language has type inference then the inferred types might contain types that the programmer did not expect. Secondly, type system features interact in complex ways and adding a new feature to the type system may result in incidental complexity on top of existing features. For example, type systems mixing parametric polymorphism and subtyping must deal with variance issues which don't exist in type systems without both features.

Prototypes and other small programs

Dynamically-typed *scripting languages* [3] have proven themselves to be extremely popular for writing short disposable programs and prototypes. This is party due to their simplicity and ease of embedding but also because in these problems domains static typing is not as beneficial. For short scripts that are constantly being changed flexibility is at a premium an runtime efficiency tends to not be as important. Additionally, there is less need for compile-time error detection when programs are small and easy to understand.

Heterogeneous data structures

In statically-typed language, the only way to package heterogeneous data into a single list is to homogenize it by using variant records. In a dynamic language, this is not necessary, since all values are already internally tagged. This means that dynamic languages are naturally suited for manipulating heterogeneous data structures, such as the following JSON [4] document:

```
{
  "name":{
      "first": "John",
      "last": "Smith"
  },
  "age": 25
}
```

In Javascript we can take advantage of the structure of JSON being similar to the structure of Javascript's (dynamically-typed) objects. For example, we can obtain John Smith's full name as follows:

```
var p = JSON.parse(...);
p.name.first + " " + p.name.last
```

The equivalent program in a statically typed language would be more verbose. For example, the same operation in Java might require the programmer to explicitly mention the type of every field [5].

```
JSONObject p = new JSONObject(...);
p.getJSONObject("name").getString("first") + " "
    + p.getJSONObject("name").getString("last")
```

Introspection

Dynamic languages make it easy to access type information at runtime. One example where this is useful is for generic traversal of data structures. For instance, the following Lua function performs a deep copy of its input, which can be a value of any type.

```
function clone(obj)
  if type(obj) == "table" then
    local copy = {}
    for k,v in pairs(obj) do
      copy[k] = clone(v)
    end
    return copy
  else
    return obj
  end
end
```

In a static language, it is harder to write these generic functions and the programmer may have to resort to creating a separate `copy` function for each input type, perhaps via some metaprogramming system [6].

2 Lambda Calculus and Type Systems

One common technique in the study of programming languages is to translate a larger language that we are interested in studying into a smaller *core language* that is easier to reason about. We will be using variations of Church's λ-calculus as the core language for many of the languages and type systems that we survey so in this chapter we provide a short will review the necessary concepts from the λ-calculus that we will use. This material on the lambda calculus and type theory is already covered in most introductory books on programming languages or type systems, such as Pierce's Types and Programming Languages [7] and the main reason we present it here is to clarify the notation we will be using. We also take the opportunity to view dynamic typing in a more formal type-theoretic setting.

2.1 The Pure Lambda Calculus

The pure λ-calculus is the simplest version of the λ-calculus and is the foundation we will use when we define our own programming languages in this text. It is an untyped functional programming language where all computation is represented as a combination of function definitions and applications.

The λ-calculus consists of two parts. Firstly, it has a syntax that specifies the language of λ-calculus programs. Secondly, it has a set of rules that describe the computational process of reducing λ-terms into other λ-terms. In Fig. 1 we present the syntax for the pure λ-calculus and a pair of operational semantics for it. We will explain these in detail in the subsections that follow.

2.1.1 Syntax

Expressions in the pure λ-calculus (also known as *λ-terms*) come in three varieties:

1. Variables, represented by letters such as x, y and z.
2. Lambda abstractions (also known as functions) are written $\boldsymbol{\lambda} x.\, e$ and abstract over a sub-expression e by parameterizing it over the variable x.
3. Function applications, written $(e_1\ e_2)$, represent the use of a function abstraction.

To avoid excessive nesting of parenthesis, it is customary to treat function application as a left-associative operation, with $(x\ y\ z)$ being equivalent to $((x\ y)\ z)$. It is also common to omit lambdas from nested function definitions. For instance, we can write $\boldsymbol{\lambda} x\, y.e$ instead of $\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} y.\, e$.

Syntax

Variable	x	$::=$	$x, y, z, \dots$
Expression	e	$::=$	$x \mid \boldsymbol{\lambda} x.\, e \mid (e_1\ e_2)$

Free Variables $\boxed{FV(e)}$

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\boldsymbol{\lambda} x.\, e) &= FV(e) - \{x\} \\ FV((e_1\ e_2)) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

Variable substitution $\boxed{e[x \leftarrow v]}$

$$\begin{aligned} x[x \leftarrow v] &= v \\ y[x \leftarrow v] &= y && \text{if } x \neq y \\ (\boldsymbol{\lambda} x.\, e)[x \leftarrow v] &= (\boldsymbol{\lambda} x.\, e) \\ (\boldsymbol{\lambda} y.\, e)[x \leftarrow v] &= (\boldsymbol{\lambda} y.\, e[x \leftarrow v]) && \text{if } x \neq y \text{ and } y \notin FV(v) \\ (e_1\ e_2)[x \leftarrow v] &= (e_1[x \leftarrow v]\ e_2[x \leftarrow v]) \end{aligned}$$

Irreducible expressions

Value v $::=$ $\boldsymbol{\lambda} x.\, e$

Reduction contexts

$C \ ::= \ [\ ] \mid (C\ e) \mid (v\ C)$

Single-step reduction $\boxed{e \longmapsto e}$

$$C[((\boldsymbol{\lambda} x.\, e)\ v)] \longmapsto C[e[x \leftarrow v]] \qquad (\textsc{app})$$

Multi-step reduction $\boxed{e \longmapsto^* e}$

$\longmapsto^*$ is the transitive and reflexive closure of $\longmapsto$

Big-step call-by-value semantics $\boxed{e \Downarrow v}$

$$\overset{\textsc{val}}{v \Downarrow v} \qquad \overset{\textsc{app}}{\dfrac{e_1 \Downarrow \boldsymbol{\lambda} x.\, e_3 \qquad e_2 \Downarrow v \qquad e_3[x \leftarrow v] \Downarrow v'}{(e_1\ e_2) \Downarrow v'}}$$

Figure 1 – The pure λ-calculus

2.1.2 Variable Bindings

Each λ-abstraction introduces a new variable name and that name can be used anywhere inside the body of the λ-abstraction. Variables without an outer function binding their name are known as *free variables*. Conversely, variables that have an outer function definition introducing their name are known as *bound variables*. If there are two nested functions defining variables with the same name, the inner function has priority and the inner definition *shadows* the name from the outer scope. Bound variables can be renamed without changing the meaning of the program.

Let expressions

Function abstractions are the only mechanism for naming things in the pure λ-calculus. However, in many situations the following **let**-expression syntax is more readable:

$$(\textbf{let}\, x = e_1 \;\textbf{in}\, e_2) \overset{\text{def}}{=} ((\boldsymbol{\lambda} x.\, e_2)\; e_1)$$

In some typed λ-calculi, such as the polymorphic calculus of Section 3.2 these **let**-expressions are given special meaning but usually they can be considered as syntactic sugar for function applications.

2.1.3 Evaluation

The fundamental operation in the pure λ-calculus is function application. Terms of the form $((\boldsymbol{\lambda} x.\, e)\;\, v)$ are known as *reducible terms* or *redexes* and can be reduced to the term $e[x \leftarrow v]$, which is the body of the function abstraction with the argument v substituted for all instances of the parameter x. This rewrite is called β-reduction and is described formally in Fig. 1. If the parameter v contains free variables and the body of the function $\boldsymbol{\lambda} x.\, e$ contains variable definitions with the same name as the free variables in v then the variables inside the function must be renamed to avoid capturing the free variables in v.

Evaluating a λ-term consists of performing β-reductions in sequence until no more reductions can be performed. This reduction process can continue forever or it can result in a final term, which is called a *normal form*. An *evaluation strategy* determines what redexes in the program can be reduced and in what order they are reduced. In this text, all the languages that we mention will use the *call-by-value* evaluation strategy, which is used by the vast majority of "real world" programming languages.

In a call-by-value setting, there is a strict and deterministic evaluation order. A redex can only be reduced if it is an outermost redex that is not located inside any λ-abstractions and if the argument to the function application has been fully reduced to a *value*, which is a normal form that is not a free variable. In the pure λ-calculus the only values are λ-abstractions but in more complex calculi values can also be booleans, numbers, strings, and so on. We will describe the call-by-value evaluation order more precisely when we describe the small-step and big-step semantics for the λ-calculus.

Under the call-by-value strategy, a normal form can be either a value or a non-value term. If the evaluation terminates with a non-value normal

$$
\begin{array}{rlr}
((\lambda x.\,e)\ v) \longmapsto e[x \leftarrow v] & & (\textsc{app}) \\
(e_1\ e_2) \longmapsto (e_1'\ e_2) & \text{if } e_1 \longmapsto e_1' & (\textsc{left}) \\
(v\ e_2) \longmapsto (v\ e_2') & \text{if } e_2 \longmapsto e_2' & (\textsc{right})
\end{array}
$$

Figure 2 – Single-step reduction rules for the λ-calculus

form we say that it has gotten *stuck*. In the pure λ-calculus, the only way a term can be stuck is if it contains free variables. For example, the function application $(x\ y)$ cannot be evaluated any further if x and y are free. However, this kind of stuck term is not very interesting because we could restrict evaluation to terms with no free variables (which are called *closed terms*) or just provide semantic meaning to free variables (perhaps by treating them as global constants). However, in richer lambda-calculi with more primitive types (integers, booleans, etc) or operations (other than function application) there might be other kinds of terms that can get stuck. One of the larger questions one can ask about a language semantics or type system is whether it is possible for programs to get stuck. Getting stuck means that the program evaluation reached a point where the way to proceed is undefined. In practical programming languages, this sort of undefined behavior often manifests as memory violations and other dangerous or unpredictable behavior.

2.1.4 Small-Step semantics

One way to more precisely describe the semantics of a λ-calculus program is via a small-step semantics [8; 9]. The fundamental element in a small-step semantics is the single-step reduction relation $e_1 \longmapsto e_2$, which means that the λ-term e_1 reduces to the term e_2 in a single step. In a call-by-value evaluation order the $\longmapsto$ relation is a partial function. The evaluation is deterministic so either an expression e_1 has exactly one image e_2 or it has no images, due to it being a fully-reduced value or stuck term. Multi-step evaluation is denoted by $\longmapsto^*$ and is defined to be the transitive and reflexive closure of $\longmapsto$. In other words, $e_1 \longmapsto^* e_2$ if and only if e_1 reduces to e_2 in zero or more steps.

The most direct way to define the $\longmapsto$ relation for the pure call-by-value λ-calculus is via the evaluation rules in Fig. 2. Rule app is the β-reduction transformation with the restriction that the argument has already been fully reduced to a value. The remaining rules describe where reductions can occur in a λ-expression. The presence of the left and right rules and the absence of other rules permitting evaluation inside of λ-abstractions results in a call-by-value evaluation order.

The reduction-context notation

Out of the three rules in Fig. 2 that we used to describe the $\longmapsto$ relation, only the app rule is actually doing interesting computations. The remaining rules are there to specify which redexes in the program can be reduced. For more complex extensions of the λ-calculus, it can be clearer to separate the computational reduction rules from these order of evaluation rules and one way to do that is via Felleisen and Hieb's reduction-context notation [10].

A *reduction-context* C represents the part of the λ-term that surrounds a redex that can be β-reduced. A reduction context is either a hole (denoted by [ ]) or λ-term with a "hole" in it, with some additional restrictions to ensure the correct evaluation order. In the pure λ-calculus with a left-to-right call-by-value evaluation order, holes can appear either in the left operand of an application or in the right operand of an application after the left operand has already been fully reduced to a value:

$$C ::= [\ ] \mid (C\ e) \mid (v\ C)$$

We denote by $C[e]$ the operation of "filling in" the hole in a reduction context with the λ-term e. This notation allows us to write a full description of the reduction semantics of the pure λ-calculus using a single rule:

$$C[((\lambda x.\, e)\ v)] \longmapsto C[e[x \leftarrow v]] \qquad (\text{APP})$$

2.1.5 Big-Step Semantics

While small-step semantics describe the runtime behavior of programs by describing each step of the program evaluation, *big-step semantics* [11], describe the behavior of programs by directly describing what value a term evaluates to. The central part of a big-step semantics is a relation $e \Downarrow v$ that relates expressions in the λ-calculus to the value to which they evaluate. In the pure λ-calculus, this relation is a partial function: Evaluation is deterministic but for some terms it does not terminate or might get stuck.

One way to define the $\Downarrow$ relation for the call-by-value λ-calculus is via the following rules:

$$\frac{}{v \Downarrow v}\ \textsc{val} \qquad \frac{e_1 \Downarrow \lambda x.\, e_3 \qquad e_2 \Downarrow v \qquad e_3[x \leftarrow v] \Downarrow v'}{(e_1\ e_2) \Downarrow v'}\ \textsc{app}$$

The way to read these evaluation rules is that if all the judgments on top of the bar are valid then the judgment under the bar is valid. The VAL rule specifies that function abstractions evaluate to themselves and the APP rule describes how to evaluate an application after the left and right operands have been evaluated.

Big-step semantics are good for writing proofs that care about the final value that an expression evaluates to but they cannot distinguish between evaluation that does not terminate and evaluation that gets stuck in a non-value. While it is possible to extend a big-step semantics with co-inductive rules that can reason about non-termination [12], in this text we will use a small-step semantic whenever we have to reason about non-termination.

2.2 Additional Datatypes and Soundness

Until now, the only values in the λ-calculus that we presented are function abstractions. In this section we will show how to extend the λ-calculus with additional primitive datatypes, which will introduce the problem of runtime type errors, a central problem to the discussion on type systems and dynamic languages.

For simplicity, we start by restricting ourselves to extending the pure λ-calculus with the simplest datatype: the **null** datatype, which contains a single value, also called null. In the type-theory literature this type is usually called **unit** but we prefer the **null** name because it is a more familiar name in a dynamic language setting. Anyway, we will call this extended calculus λ^{NULL}. As is shown in Fig. 3, the **null** type does not have any operations associated with it so the language semantics of λ^{NULL} contains exactly the same evaluation rules as the pure λ-calculus. The only change between them is the addition of null as a valid term and that the set of irreducible values now includes null in addition to function abstractions.

This latter change in the set of irreducible values has a big consequence, however, which is why we say that the λ^{NULL} calculus is incomplete. It is now possible for the evaluation of a λ-term to get stuck on a non-value if a null value appears as the left operand of a function application.

Similar problems occur if we add any other datatype to the pure λ-calculus. **null** is just the simplest possible case. For example, had we added a datatype for booleans we would also have created the possibility of function applications getting stuck due to receiving a boolean where a function was expected or if a function were passed as the conditional of an **if-then-else** expression.

2.2.1 Soundness

In real-world programming languages, undefined behavior due to stuck terms can be a source of unpredictable program behavior, such as buffer overflows and segmentation faults, which can lead to serious bugs. It is therefore highly desirable for a programming language to not allow any instances of such undefined behavior. This is known as *soundness*. In a sound programming language, evaluating a program either terminates with a value or enters an infinite loop – it never gets stuck.

There are two major approaches for making programming languages sound. In Section 2.3 we present the approach taken in dynamic languages, which is to assign a meaning to all the previously stuck terms. This is what is done in dynamic languages. In Section 2.4 we cover the approach of statically-typed languages, which use a static type system to restrict the set of programs that can be executed to only the programs that can be guaranteed at compile-time that they will never get stuck.

2.3 A Dynamically-Typed Lambda Calculus

Dynamically-typed languages use runtime checks to detect "stuck" terms and assign an appropriate meaning to them. The simplest way to avoid having to deal with stuck terms is to cleanly abort the program execution by raising an exception. In this section we will present an example of this approach, the λ^{DYN} calculus, a dynamically-typed version of the λ^{NULL} calculus from Section 2.2.

Syntax

Variable	x	$::=$	$x, y, z, \ldots$
Function	f	$::=$	$\boldsymbol{\lambda} x.\, e$
Null		$::=$	null
Expression	e	$::=$	$x \mid f \mid \text{null} \mid (e_1\ e_2)$

Irreducible expressions

Value	v	$::=$	$f \mid \text{null}$

Reduction contexts

$$C \ ::= \ [\ ] \mid (C\ e) \mid (v\ C)$$

Single-step reduction $\boxed{e \longmapsto e}$

$$C[((\boldsymbol{\lambda} x.\, e)\ v)] \longmapsto C[e[x \leftarrow v]] \qquad (\textsc{app})$$

Figure 3 – An incomplete λ-calculus with null ($\lambda^{\textsc{null}}$)

2.3.1 Semantics of $\lambda^{\textsc{dyn}}$

The $\lambda^{\textsc{dyn}}$ calculus and its semantics are described fully in Fig. 4. The main difference compared to the $\lambda^{\textsc{null}}$ calculus is the introduction of the NOT-A-FUNCTION exception. Instead of program evaluation always resulting in a value, it can now result in either a value or an exception, which is reflected in the codomains of the $\longmapsto$ and $\longmapsto^*$ partial functions.

The APP-ERR rule deals with runtime errors. If at any point the evaluation encounters a function application that would have gotten stuck in $\lambda^{\textsc{null}}$, the evaluation immediately aborts with a NOT-A-FUNCTION exception.

From a type systems point of view, the distinctive feature of $\lambda^{\textsc{dyn}}$ is that it is a sound programming language. Evaluation never gets stuck because the exception-handling rules now assign a meaning to all the terms that would have gotten stuck in the $\lambda^{\textsc{null}}$ calculus.

2.3.2 Weak Typing

Raising a runtime exception when an operation receives inputs of an inappropriate type is the most direct way to create a sound dynamic language. However, another possibility is to assign some non-error meaning to the ill-typed terms, perhaps by coercing the inputs to a different type. These coercions blur the distinction between the different types and many programmers refer to the resulting language as a *weakly-typed* language.

One example of weak typing in a dynamic language is how in the PHP language the arithmetic operators coerce string inputs to numbers. [13]. For example, the PHP expression `"1a"+"2b"` evaluates to the number 3 because the addition converts its inputs to numbers and the conversion from strings to numbers ignores all characters after the initial numeric prefix. These

Syntax

Variable	x	::=	$x, y, z, \ldots$
Function	f	::=	$\lambda x.\, e$
Null		::=	null
Expression	e	::=	$x \mid f \mid \text{null} \mid (e_1\ e_2)$

Irreducible expressions

Value	v	::=	$\text{null} \mid f$
Error	Ω	::=	NOT-A-FUNCTION
Result	r	::=	$v \mid \Omega$

Reduction contexts

$$C \quad ::= \quad [\ ] \mid (C\ e) \mid (v\ C)$$

Single-step reduction $\boxed{e \longmapsto (e \cup \Omega)}$

$$C[((\lambda x.\, e)\ v)] \longmapsto C[e[x \leftarrow v]] \qquad (\textsc{app})$$

$$C[(v_1\ v_2)] \longmapsto \textsc{not-a-function} \qquad \text{if } v_1 \notin \text{Function} \qquad (\textsc{app-err})$$

Figure 4 – The dynamically-typed λ-calculus with null (λ^{DYN})

permissive coercions can sometimes "swallow" programmer errors and hide the true source of bugs by preventing or delaying desirable exceptions from being raised.

What this all means is that the soundness of a programming language is a relative thing. If we follow the definition to the letter, PHP is a sound programming language where programs never get stuck. However, if a programmer finds that one of the implicit coercions is a common source of bugs than that coercion can end up being as undesirable as undefined behavior would have been.

2.4 Simple Types

In this section, we introduce the simply-typed λ-calculus ($\lambda^{\rightarrow}$) and its type system. The type system for the simply-typed λ-calculus is one of the simplest type systems and it is used as a building block for many other more advanced type systems. It is also a good starting place to introduce type system terminology.

2.4.1 Syntax and Semantics

The syntax and semantics for $\lambda^{\rightarrow}$ are practically the same as the syntax and semantics for the λ^{NULL} calculus. The only difference is that the parameters in function abstractions now carry explicit type annotations in order to simplify

Type Syntax

$$\text{Type} \quad \tau \quad ::= \quad \mathbf{null} \mid \tau_1 \to \tau_2$$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{}{\Gamma \vdash \text{null} : \mathbf{null}}\ \textsc{null} \qquad \frac{\Gamma[x \leftarrow \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\boldsymbol{\lambda} x : \tau_1 . e) : (\tau_1 \to \tau_2)}\ \textsc{lam}$$

$$\frac{\Gamma \vdash e_1 : (\tau \to \tau') \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}\ \textsc{app}$$

Figure 5 – The type system of the simply-typed lambda calculus with null ($\lambda^{\to}$)

the type checking process. In a later section we will cover *type inference*, which allows these annotations to be omitted.

2.4.2 Types

Informally, the type of an expression is a computable superset of the set of values the expression may evaluate to. A type system consists of a definition of what the types can be and a set of rules for computing the type of each expression in a given program.

We show the type system for $\lambda^{\to}$ in Fig. 5. A type can be either a ground type (which in our simplified system can only mean the **null** type) or a function type $(\tau_1 \to \tau_2)$, for functions that take inputs of type τ_1 and return values of type τ_2.

A $\lambda^{\to}$ program is considered *well-typed* if it is possible to assign types to each of its sub expressions in a way that respects the typing rules shown in Fig. 5. The typing rules use *type environments* Γ, which map the names of variables in scope to their types. $\Gamma(x)$ is the type of variable x in the environment Γ and $\Gamma[x \leftarrow \tau]$ is the environment obtained by extending the environment Γ with a variable x of type τ. Typing judgments take the form $\Gamma \vdash e : \tau$ and mean that in a type environment Γ the expression e has type τ.

Examples

First, let us show an example of an well-typed program. Consider the following program:

$$((\boldsymbol{\lambda} x : \mathbf{null} . x)\ \text{null})$$

This program is well typed and can be assigned the type **null**, as is shown by the following type derivation tree:

$$\cfrac{\cfrac{\cfrac{}{\{x : \mathbf{null}\} \vdash x : \mathbf{null}}\ \textsc{var}}{\{\} \vdash (\boldsymbol{\lambda} x : \mathbf{null} . x) : (\mathbf{null} \to \mathbf{null})}\ \textsc{lam} \qquad \cfrac{}{\{\} \vdash \text{null} : \mathbf{null}}\ \textsc{null}}{\{\} \vdash ((\boldsymbol{\lambda} x : \mathbf{null} . x)\ \text{true}) : \mathbf{null}}\ \textsc{app}$$

For an example of a program that cannot be typed in the $\lambda^{\rightarrow}$ calculus consider the following:

$$(\text{null null})$$

This program cannot be assigned a type because the only rule that applies to function applications is the APP rule and that rule cannot be used for this program because it requires the left operand to have a function type and null is not a function.

2.4.3 Soundness

It is not a coincidence that the expression (null null) is ill-typed, as the type system for the simply-typed λ-calculus was designed to be sound. All terms that get stuck during evaluation are ill-typed or, as the popular saying goes, well-typed programs do not go wrong (get stuck).

For the dynamically-typed λ-calculus in Section 2.3 it was easy to directly prove the soundness of the language. For languages with multiple types, such as $\lambda^{\rightarrow}$, this process is slightly more complicated. Soundness is usually proved in two steps, by showing that the language has the *progress* property and the *preservation* property:

Definition (Progress)**.** A well-typed expression e_1 is either a fully reduced value or it can be further reduced to another expression e_2. The evaluation does not get immediately stuck.

Definition (Preservation)**.** If a well-typed expression e_1 reduces to another expression e_2 then e_2 is also well typed and has the same type as e_1.

A full proof of the soundness of the simply-typed λ-calculus can be found in Pierce's Types and Programming Languages [7] or any other book on type systems. For brevity we include here only a proof sketch:

The proof of the progress property involves showing that all stuck terms are ill-typed, which is the contrapositive of the progress property. There is only one kind of term in λ^{NULL} that gets stuck: function applications with a null value as the left operand. Since none of the typing rules in the simply-typed calculus apply to these stuck terms, they cannot be well-typed.

The proof of preservation can be done via an exhaustive case analysis over the reduction rules of the $\lambda^{\rightarrow}$ and is left as an exercise for the reader.

2.4.4 Type Inference

So far, we have required explicit type annotations for all variable declarations in the simply-typed λ-calculus. However, this can be very burdensome and can also get in the way of using type systems to reason about dynamic languages, where programs do not have any type annotations. The alternative to using explicit type signatures is to use a *type inferencing* algorithm to determine what types could fit the missing type annotations.

For the simply-typed λ-calculus, type inference can be formulated as finding a solution to a system of equations involving type variables. Each variable x and each program subexpression e in the program is associated

Phrase	Constraints
null	$[\![\text{null}]\!] = \textbf{null}$
$\boldsymbol{\lambda} x.\, e$	$[\![\boldsymbol{\lambda} x.\, e]\!] = [\![x]\!] \to [\![e]\!]$
$(e_1\ e_2)$	$[\![e_1]\!] = [\![e_2]\!] \to [\![(e_1\ e_2)]\!]$

Figure 6 – Type inference constraints for the simply-typed λ-calculus

with a *type variable*, which we will represent by $[\![x]\!]$ and $[\![e]\!]$, respectively. The typing rules for the simply-typed λ-calculus impose a set of equality restrictions between the types of different parts of the program. Each syntactic form in the program leads to a different set of restrictions, as summarized in Fig. 6. Type inference then consists of searching for a solution to the set of constraints. One way to do this is via an unification algorithm. Its more common to see this unification algorithm being presented for polymorphic λ-calculi but a version specialized for the simply-typed calculus can be found in Oleg Kiselyov's class notes [14].

2.5 Type System Conservativeness and "Acceptable Losses"

While type systems guarantee that well-typed programs don't go wrong, they do not guarantee the converse, that ill-typed programs will go wrong. Whether a program gets stuck or not is a nontrivial runtime property, which is undecidable according to Rice's theorem [15]. Because of this undecidability, any type system must err on the side of conservativeness and statically reject some programs that would have successfully executed had the type system not prevented them from being executed in the first place.

It is easy to come up with trivial example programs that do not get stuck but which will be rejected by most type systems. For example, in the following program the type error occurs inside a branch of the **if-then-else** expression that never gets executed. [1]

$$\textbf{if}\ \text{true}\ \textbf{then}\ 10\ \textbf{else}\ (\text{null null})$$

However, examples like this one are not the most interesting. Dead branches are a code smell and usually are not added to a program intentionally. A bigger problem occurs when the type system cannot type a *useful* program that the programmer would have liked to be able to write.

A prime example of a type system rejecting useful programs is the fact that the simply-typed λ-calculus, as we presented in Section 2.4, is not Turing-complete [16]. Programs that use recursion are not just hard to write in the simply-typed λ-calculus, they are impossible to write!

Another, less extreme, example of a type system limitation is the lack of polymorphism in the simply-typed λ-calculus. Consider the following

[1] We formally describe **if-then-else** expressions in Section 2.8. For now the reader can assume that booleans and conditionals behave as expected

program which consists of a single function being applied to values of different types:

$$\begin{aligned}&\textbf{let} f = (\lambda x.\, x)\ \textbf{in}\\ &\textbf{if}\ (f\ \text{true})\ \textbf{then}\ (f\ 5)\ \textbf{else}\ (f\ 7)\end{aligned}$$

It successfully evaluates to 5 but it cannot be given a type in the simply-typed λ-calculus because the f function cannot be typed **bool** → **bool** and **int** → **int** simultaneously. While it is possible to write an equivalent program that can be typed using only simple types, doing so requires giving up f's code reuse:

$$\begin{aligned}&\textbf{let} f = (\lambda x.\, x)\ \textbf{in}\\ &\textbf{let}\, g = (\lambda x.\, x)\ \textbf{in}\\ &\textbf{if}\ (f\ \text{true})\ \textbf{then}\ (g\ 5)\ \textbf{else}\ (g\ 7)\end{aligned}$$

Given the advantages that type systems provide, the useful programs that a typed language unable to express may be seen as *acceptable losses* in a complex tradeoff. On the other hand, expressiveness and flexibility of the programming language are highly valued by programmers so one of the biggest endeavors of type-system research is developing expressive type-systems that are able to type more useful programming patterns. For instance, in Section 2.6 we show how to enrich the simply-typed λ-calculus so it can support recursive functions and in Section 3.2 we will show a polymorphic type system that solves the second limitation that we mentioned.

Changing topics a bit, the inherent expressiveness limitations of statically are a compelling argument for dynamic languages. Dynamic languages sacrifice static program checking for the freedom to execute every syntactically valid program. The search for ways to benefit from both the expressiveness of dynamic languages and the static guarantees of typed languages is the driving force for research on type systems for dynamic languages, which is the focus of this text.

2.6 Recursion

The pure λ-calculus is Turing-complete and it is possible to express recursive functions by using fixed-point combinators. For example, the following figure defines a factorial function FAC for a λ-calculus that has been extended with numbers and arithmetic operations.

$$\begin{aligned}Z &= \lambda f.\, (\lambda x. f\ (\lambda v.\, ((x\ x)\ v)))\ (\lambda x. f\ (\lambda v.\, ((x\ x)\ v)))\\ \text{FAC} &= (Z\ (\lambda f.\, \lambda n.\, \textbf{if}\ (n = 0)\ \textbf{then}\ 1\ \textbf{else}\ (n \times (f\ (n-1)))))\end{aligned}$$

However, these fixed point combinators are very inconvenient to use and they also cannot be typed in the simply-typed λ-calculus and many other type systems. Because of this, it can be useful to add explicit support for recursion to the language.

One approach is to add a fixed point operator **fix** as a language primitive, as is shown in Fig. 7. In the expression ($\textbf{fix} f.\, e$), the body e can refer to itself

Additional syntactic forms

$$\text{Expr} \quad e \quad ::= \quad \ldots \mid (\mathbf{fix} f.\, e)$$

Additional small-step semantics rules

$$C[(\mathbf{fix} f.\, e)] \longmapsto C[e[f \leftarrow (\mathbf{fix} f.\, e)]] \qquad (\textsc{fix})$$

Additional typing rules

$$\textsc{fix}\quad \frac{\Gamma[f \leftarrow \tau] \vdash e : \tau}{\Gamma \vdash (\mathbf{fix} f.\, e) : \tau}$$

Figure 7 – Adding recursion to the λ-calculus

via the variable f. This can be seen in the FIX evaluation rule, which reduced $(\mathbf{fix} f.\, e)$ into $e[f \leftarrow (\mathbf{fix} f.\, e)]$.

The inclusion of the **fix** operator does not introduce any new runtime errors (stuck terms), since evaluating a **fix** expression always succeeds. Because of this, the typing rule for **fix** is not surprising.

Using **fix**, it is possible to define the factorial function from before as follows:

$$\text{FAC} = (\mathbf{fix} f.\, \lambda n.\, \mathbf{if}\ (n = 0)\ \mathbf{then}\ 1\ \mathbf{else}\ (n \times (f\ (n - 1)))))$$

Letrec expressions

Just like **let**-expressions can be easier to read than function applications, it is often clearer to define recursive functions with a **letrec** syntax instead of using **fix** directly.

$$(\mathbf{letrec} f = \lambda x.\, e_1\ \mathbf{in}\ e_2) \overset{\text{def}}{=} (\mathbf{let} f = (\mathbf{fix} f.\, \lambda x.\, e_1)\ \mathbf{in}\ e_2)$$

2.7 Products and Records

A very important programming language feature is the ability to define custom datatypes that aggregate multiple values in a single value that can be passed around as a unit. The most general form of this are records, which are a collection of named fields. However, aiming for simplicity, we will describe in more detail the specific case of the *pair* datatype, which is also known as a *product*.

In figure 8 we show an extension of the pure λ-calculus with a pair datatype. The additions to the syntax are a **pair** function to create new pairs and the **fst** and **snd** functions to extract values from pairs. The set of irreducible values now also includes pairs of values in addition to the existing functions and booleans. The additional evaluation rules are the FST and SND rules to specify how to extract a value from a pair.

$$
\begin{array}{llll}
\text{Variable} & x & ::= & x, y, z, \ldots \\
\text{Function} & f & ::= & \boldsymbol{\lambda} x.\, e \\
\text{Null} & & ::= & \text{null} \\
\text{Pair} & p & ::= & (\textbf{pair}\ e_1\ e_2) \\
\text{Expression} & e & ::= & x \mid f \mid \text{null} \mid (e_1\ e_2) \mid (\textbf{fst}\ e) \mid (\textbf{snd}\ e)
\end{array}
$$

Irreducible expressions

$$
\begin{array}{llll}
\text{Value} & v & ::= & f \mid \text{null} \mid (\textbf{pair}\ v_1\ v_2)
\end{array}
$$

Small-step call-by-value semantics

Reduction contexts
$C \quad ::= \quad [\ ] \mid (C\ e) \mid (v\ C) \mid (\textbf{pair}\ C\ e) \mid (\textbf{pair}\ v\ C)$
Single-step reduction $\boxed{e \longmapsto e}$

$$
\begin{array}{rll}
C[((\boldsymbol{\lambda} x.\, e)\ v)] & \longmapsto C[e[x \leftarrow v]] & (\textsc{app}) \\
C[(\textbf{fst}\ (\textbf{pair}\ v_1\ v_2))] & \longmapsto C[v_1] & (\textsc{fst}) \\
C[(\textbf{fst}\ (\textbf{pair}\ v_1\ v_2))] & \longmapsto C[v_2] & (\textsc{snd})
\end{array}
$$

Figure 8 – An incomplete λ-calculus with pairs

Pairs are a new datatype that we added to the λ-calculus so they will give rise to type errors similarly to how the addition of a **null** datatype did. Again, we can return the calculus to soundness via a dynamic or a static approach.

2.7.1 Dynamic error handling

Handling pair errors dynamically is a matter of adding a NOT-A-PAIR error to the set of possible exceptions and adding the appropriate rules to the language semantics:

$$
\begin{array}{rlll}
C[(\textbf{fst}\ v)] & \longmapsto \textsc{not-a-pair} & \text{if } v \notin \text{Pair} & (\textsc{fst-err}) \\
C[(\textbf{snd}\ v)] & \longmapsto \textsc{not-a-pair} & \text{if } v \notin \text{Pair} & (\textsc{snd-err})
\end{array}
$$

2.7.2 Static type checking

To type pairs statically we extend the type language with *pair-types*, which have the form $\tau_1 \times \tau_2$. The typing rules for pair types are shown in Fig. 9.

2.7.3 Records

Records are a more powerful version of pairs. Instead of only storing two elements, records can store any number of them. Each element is stored in a *field*, which is identified by an unique *label*. The record notation we use in this text consists of a list of fields enclosed in braces. For example, the following record represents a point in three-dimensional space.

$$\{x\!:\!10,\ y\!:\!20,\ z\!:\!30\}$$

Type Syntax

$$\text{Type} \quad \tau \quad ::= \quad \mathbf{null} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\textsc{pair}\;\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\mathbf{pair}\; e_1\; e_2) : (\tau_1 \times \tau_2)} \qquad \textsc{fst}\;\frac{\Gamma \vdash e : (\tau_1 \times \tau_2)}{\Gamma \vdash (\mathbf{fst}\; e) : \tau_1} \qquad \textsc{snd}\;\frac{\Gamma \vdash e : (\tau_1 \times \tau_2)}{\Gamma \vdash (\mathbf{snd}\; e) : \tau_2}$$

Figure 9 – Typing rules for pair types

The order of the fields does not matter. Accessing a field is done via dot notation:

$$\mathbf{let}\; r = \{x\!:\!10,\; y\!:\!20,\; z\!:\!30\}\; \mathbf{in}\; r.x$$

Our syntax for record types is similar to the syntax for record values. The type of the three-dimensional points we used as an example is:

$$\{x\!:\!\mathbf{int},\; y\!:\!\mathbf{int},\; z\!:\!\mathbf{int}\}$$

2.8 Sums and Variants

Sums and variants are the dual versions of products and records. While a product contains a pair of values both at once, a sum contains one of a pair of possible values each time.

In figure 10 we show an extension of the pure λ-calculus with a sum datatype. The additions to the syntax are a pair of tagging functions **left** and **right** and **case** expressions to consume these tagged values. The set of irreducible values now also includes tagged values in addition to the existing functions and booleans. The additional evaluation rules are the CASE-LEFT and CASE-RIGHT rules, which specify how to *pattern-match* over a tagged value.

Sums are a new datatype that we added to the λ-calculus so they will give rise to type errors similarly to how the addition of a **null** datatype and of pairs did. Again, we can return the calculus to soundness via a dynamic or a static approach.

2.8.1 Dynamic error handling

Handling pattern-matching errors dynamically is a matter of adding a NOT-A-TAG error to the set of possible exceptions and adding the appropriate rules to the language semantics:

$$C[(\mathbf{case}\; v\; \mathbf{of}\; (\mathbf{left}\; x) \Rightarrow e_1\; ; (\mathbf{right}\; y) \Rightarrow e_1)] \longmapsto \textsc{not-a-tag} \qquad \text{if } v \notin \text{Tag} \qquad (\textsc{case-err})$$

2.8.2 Static type checking

To type sums statically we extend the type language with *sum-types*, which have the form $\tau_1 + \tau_2$. The typing rules for pair types are shown in Fig. 11.

Variable	x	$::=$	$x, y, z, \ldots$
Function	f	$::=$	$\boldsymbol{\lambda} x.\, e$
Null		$::=$	null
Tag	tag	$::=$	$(\textbf{left}\ e) \mid (\textbf{right}\ e)$
Expression	e	$::=$	$x \mid f \mid \text{null} \mid tag \mid (e_1\ e_2) \mid$
			$(\textbf{case}\ e_1\ \textbf{of}\ (\textbf{left}\ x) \Rightarrow e_2\ ; (\textbf{right}\ y) \Rightarrow e_3)$

Irreducible expressions

Value v $::=$ $f \mid \text{null} \mid (\textbf{left}\ v) \mid (\textbf{right}\ v)$

Small-step call-by-value semantics

Reduction contexts

$C \ ::= \ [\ ] \mid (C\ e) \mid (v\ C) \mid (\textbf{case}\ C\ \textbf{of}\ (\textbf{left}\ x) \Rightarrow e_1\ ; (\textbf{right}\ y) \Rightarrow e_2)$

Single-step reduction $\boxed{e \longmapsto e}$

$$C[((\boldsymbol{\lambda} x.\, e)\ v)] \longmapsto C[e[x \leftarrow v]] \quad (\textsc{app})$$

$$C[(\textbf{case}\ (\textbf{left}\ v)\ \textbf{of}\ (\textbf{left}\ x) \Rightarrow e_1\ ; (\textbf{right}\ y) \Rightarrow e_2)] \longmapsto C[e_1[x \leftarrow v]] \quad (\textsc{case-left})$$

$$C[(\textbf{case}\ (\textbf{right}\ v)\ \textbf{of}\ (\textbf{left}\ x) \Rightarrow e_1\ ; (\textbf{right}\ y) \Rightarrow e_2)] \longmapsto C[e_2[y \leftarrow v]] \quad (\textsc{case-right})$$

Figure 10 – An incomplete λ-calculus with sums

Type Syntax

Type τ $::=$ $\textbf{null} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\textsc{left}\quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (\textbf{left}\ e) : (\tau_1 + \tau_2)} \qquad \textsc{right}\quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash (\textbf{right}\ e) : (\tau_1 + \tau_2)}$$

$$\textsc{case}\quad \frac{\Gamma \vdash e_1 : (\tau_1 + \tau_2) \qquad \Gamma[x \leftarrow \tau_1] \vdash e_2 : \tau' \qquad \Gamma[y \leftarrow \tau_2] \vdash e_3 : \tau'}{\Gamma \vdash (\textbf{case}\ e_1\ \textbf{of}\ (\textbf{left}\ x) \Rightarrow e_2\ ; (\textbf{right}\ y) \Rightarrow e_3) : \tau'}$$

Figure 11 – Typing rules for sum types

2.8.3 Variants

Just like we informalliy presented records as a generalization of products, we will introduce a gereralization of sums with named fields. A *variant* type consists of a list of tag names and their associated data. We will denote variant types between angle brackets and separated by "|", to indicate that the different cases are exclusive. For a concrete example, the following variant type can be used as the return type of a function that can either succeed or fail with an error message:

$$<\text{Success}:\mathbf{unit} \mid \text{Error}:\mathbf{string}>$$

2.8.4 Booleans

A very important particular case of variant types is the boolean type. Most programming languages support booleans and **if**-**then**-**else** expressions even if they do not offer support for general variants. It is possible to encode booleans as variants as follows:

$$\mathbf{bool} \overset{\text{def}}{=} <\text{TagTrue}:\mathbf{unit} \mid \text{TagFalse}:\text{unit}>$$

$$\begin{aligned} \text{true} &\overset{\text{def}}{=} (\text{TagTrue unit}) \\ \text{false} &\overset{\text{def}}{=} (\text{TagFalse unit}) \\ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 &\overset{\text{def}}{=} (\mathbf{case}\ e_1\ \mathbf{of}\ (\text{TagTrue}\ \_) \Rightarrow e_2\ ; (\text{TagFalse}\ \_) \Rightarrow e_3) \end{aligned}$$

3 Advanced Type Systems

In Chapter 2 we introduced the λ-calculus and many basic extensions for it, including records and fix-point operators. For each feature that we introduced, we also explained the simplest type system that can be used to integrate that feature into the λ-calculus.

In this chapter we will cover type system features that are commonly used in the literature for types for dynamic languages but that do not require extensions to the evaluation rules of the underlying λ-calculus. These type system features make the typed languages more expressive by allowing things that could not be typed before to be typed or by making it possible to create types that are more specific than before.

In Section 3.1 we present recursive types, which make it possible to type some data structures that were not typable before. Section 3.2 is about parametric polymorphism, which allows reusing functions that are indifferent to the types of their inputs in many different contexts. In Section 3.3 we present subtyping, which allows reasoning about types as subsets of other types, as well as making it possible to implicitly convert from a more specific type to a more general type. Section 3.4 covers union types, which in dynamic languages are often a more idiomatic alternative to variant types. Finally, in Section 3.5 we describe a way to represent finitary overloading via intersection types.

3.1 Recursive Types

Some datatypes are defined inductively. For example, the set of Peano numerals is defined to be the smallest set that contains the number zero and the successors of every numeral.

Adding recursion on the type level allows type system to give a finite representation for the type of these inductively-defined data sets. The syntax we will use for recursive types will be the type-level fix-point operator μ. The type $\mu\alpha.\tau$ is a recursive type that can refer to itself via the α type variable.

For example, using the variant type notation from Section 2.8 we can represent the type of Peano numerals as follows:

$$\mu N.<\mathrm{zero}:\mathrm{unit} \mid \mathrm{succ}:N>$$

Formally giving meaning to these recursive types is subtle. There are two different approaches to do so: *equi-recursive* types and *iso-recursive* types.

In an equi-recursive setting, the recursive type $\mu\alpha.\tau$ and its unfolding $\tau[\alpha \leftarrow \mu\alpha.\tau]$ are considered to be equivalent and completely interchangeable. In other words, the following typing rules apply:

$$\text{FOLD}\quad \frac{\Gamma \vdash e : \tau[\alpha \leftarrow \mu\alpha.\tau]}{\Gamma \vdash e : \mu\alpha.\tau} \qquad\qquad \text{UNFOLD}\quad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \mu\alpha.\tau]}$$

Additional syntax

Expr	e	::=	$\ldots \mid (\textbf{fold}\ e) \mid (\textbf{unfold}\ e)$
Value	v	::=	$\ldots \mid (\textbf{fold}\ e)$

Changes to reduction semantics

Reduction contexts
$C \quad ::= \quad ... \mid (\textbf{fold}\ C) \mid (\textbf{unfold}\ C)$
Evaluation rules

$$C[(\textbf{unfold}\ (\textbf{fold}\ e))] \longmapsto C[e] \qquad (\textsc{unfold-fold})$$

Additional typing rules

$$\textsc{fold}\ \frac{\Gamma \vdash e : \tau[\alpha \leftarrow \mu\alpha.\tau]}{\Gamma \vdash (\textbf{fold}\ e) : \mu\alpha.\tau} \qquad \textsc{unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash (\textbf{unfold}\ e) : \tau[\alpha \leftarrow \mu\alpha.\tau]}$$

Figure 12 – Typing rules for iso-recursive types

While equi-recursive types are very simple, the typing rules are not syntax-directed, which can make type checking and type inferencing considerably more difficult.

In an iso-recursive setting, the recursive type $\mu\alpha.\tau$ and its unfolding $\tau[\alpha \leftarrow \mu\alpha.\tau]$ are not considered to be equal and interchangeable. They are just isomorphic and the programmer must explicitly use a conversion function to convert from one form to the other. In Fig. 12 we describe the changes necessary to add iso-recursive recursive types to the simply-typed λ-calculus. Two new syntactic forms are added to the language, the **fold** and **unfold** annotations, ac can be seen in the evaluation rules. These annotations do not have any computational meaning and exist solely to guide the type checker. Unlike what happens in equi-recursive systems, the presence of the **fold** and **unfold** annotations allows type checking to be syntax-directed.

3.1.1 Iso- vs Equi-recursive Types

The majority of statically-typed programming languages with recursive types use iso-recursion instead of equi-recursion. Type checking and inference in an iso-recursive setting is much simpler and type error messages can be more predictable. At the same time, the downside of needing to insert **fold** and **unfold** annotations is often not a problem in practice. For example, in ML and related languages each use of a constructor for a recursive type implicitly introduces a **fold** and pattern matching implicitly introduces the complementary **unfold**s. Therefore, the programmer does not need to actually introduce additional annotations to the program in order to use recursive types.

However, in programming languages with no type annotations or explicit constructor and tagging functions, equi-recursion may be the only feasible alternative. We will see an example of this in Section 4.1, where a type system for a dynamic language uses equi-recursion.

3.1.2 Embedding Dynamic Typing in a Typed Language

One very important type that can be represented with type recursion is the variant type for dynamically-typed values. For example, in a language with booleans and integers as basic types, the following variant can represent any possible value:

$$\mu D. <\mathrm{bool}\!:\!\mathbf{bool} \mid \mathrm{int}\!:\!\mathbf{int} \mid \mathrm{func}\!:\!D \rightarrow D>$$

Because of this, some type theorists say that untyped languages are actually-typed and just happen to have a single type that contains every value. This "untyped languages are actually uni-typed" motto was popularized by Robert Harper and is attributed to Dana Scott [17].

3.2 Universal Types

As we previously hinted in Section 2.5, some expressions in the simply-typed λ-calculus can be assigned more than one type. For example, the identity function $\lambda x.\, x$ is indifferent to the type of its input and can be assigned any type that has the form $\tau \rightarrow \tau$. One of the ways we can extend the simply-typed λ-calculus to make it more expressive is to bring these higher-level type-schemes into the type system.

Allowing a single program expression to be used with different types in different parts of the program is known as *polymorphism*. One form of polymorphism is *parametrically polymorphism*, where types can be parameterized by type variables. For example, the polymorphic type of the identity function can be written as $\forall \alpha . \alpha \rightarrow \alpha$.

In Fig. 13 we describe these *universal types* in more detail. In addition to the ground types and function types from the simply-typed λ-calculus, a calculus with parametric polymorphism also features type variables, usually represented with Greek letters like α and β, and universally quantified types, like $\forall \alpha . \tau$. Similarly to how regular variables are bound in λ-abstractions, type variables are bound in ∀-quantifiers. We denote by $\mathrm{FTV}(\tau)$ the set of free type variables in the type τ. Another similarity type variables have with regular variables is the substitution operation, which we represent by $\tau_1[\alpha \leftarrow \tau_2]$ and works by substituting τ_2 for all free occurrences of α in τ_1.

3.2.1 Let Polymorphism

Extending the simply-typed λ-calculus with the most general version of universal types results in a calculus known as System F [18]. However, type checking and type inference for System F are undecidable [19] so practical versions of λ-calculi with universal types add some restrictions to how the universal types can be used.

Type Variable	α	$::=$	$\alpha, \beta, \ldots$
Type	τ, σ	$::=$	$\mathbf{bool} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha . \tau$

Free type variables $\boxed{\mathrm{FTV}(\tau)}$

$$\begin{aligned}
\mathrm{FTV}(\alpha) &= \{\alpha\} \\
\mathrm{FTV}(\mathbf{bool}) &= \{\} \\
\mathrm{FTV}(\tau_1 \to \tau_2) &= \mathrm{FTV}(\tau_1) \cup \mathrm{FTV}(\tau_2) \\
\mathrm{FTV}(\forall \alpha . \tau) &= \mathrm{FTV}(\tau) - \{\alpha\}
\end{aligned}$$

Type substitution $\boxed{\tau[\alpha \leftarrow \tau]}$

$$\begin{aligned}
\alpha[\alpha \leftarrow \tau] &= \tau \\
\beta[\alpha \leftarrow \tau] &= \beta && \alpha \neq \beta \\
\mathbf{bool}[\alpha \leftarrow \tau] &= \mathbf{bool} \\
(\tau_1 \to \tau_2)[\alpha \leftarrow \tau] &= \tau_1[\alpha \leftarrow \tau] \cup \tau_2[\alpha \leftarrow \tau] \\
(\forall \alpha . \sigma)[\alpha \leftarrow \tau] &= (\forall \alpha . \sigma) \\
(\forall \beta . \sigma)[\alpha \leftarrow \tau] &= \forall \beta . (\sigma[\alpha \leftarrow \tau]) && \beta \in \mathrm{FTV}(\tau)
\end{aligned}$$

Figure 13 – Type syntax for general universal types

One of these restrictions is Milner's *let polymorphism* [20]. In a let-polymorphic setting, types are divided between *monotypes*, which contain no type quantifiers and *polytypes*, which abstract over a monotype via one or more type variables. The main restriction in a let-polymorphic system and reason for the separation between monotypes and polytypes is that type variables can only be instantiated to monotypes.

The let-polymorphic calculus extends the syntax of the λ-calculus with **let** expressions, which will be the only way to introduce polymorphically-typed identifiers. These expressions are the reason for the name "let-polymorphism".

$$\mathbf{let} f = \lambda x.\, x \;\mathbf{in}\; \mathbf{if}\; (f\ \mathrm{true})\; \mathbf{then}\; (f\ 5)\; \mathbf{else}\; (f\ 7)$$

3.2.2 Typing Rules

Typing judgments in the let-polymorphic λ-calculus have the form $\Gamma \vdash e : \tau$ and mean that in a type environment Γ the expression e has (mono)type t. Type environments map program variable names to their (poly)types.

We show the typing rules for the let-polymorphic λ-calculus in Fig. 14. There are two groups of typing rules. The first group consists of the BOOL, IF, LAM and APP rules, which are the same as in a simply-typed λ-calculus. The remaining LET and VAR rules are responsible for the type polymorphism.

Monotypes and polytypes

Type Variable	α	::=	$\alpha, \beta, \ldots$
Monotype	τ	::=	$\mathbf{bool} \mid \tau_1 \to \tau_2 \mid \alpha$
Polytype	σ	::=	$\forall \alpha_1 \ldots \forall \alpha_n . \tau$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\textsc{var}\quad \frac{\Gamma(x) = \sigma \qquad \tau \sqsubseteq \sigma}{\Gamma \vdash x : \tau} \qquad \textsc{bool}\quad \frac{b \in \text{Boolean}}{\Gamma \vdash b : \mathbf{bool}} \qquad \textsc{if}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{bool} \\ \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau\end{array}}{\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) : \tau}$$

$$\textsc{lam}\quad \frac{\Gamma[x \leftarrow \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : (\tau_1 \to \tau_2)} \qquad \textsc{app}\quad \frac{\Gamma \vdash e_1 : (\tau \to \tau') \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}$$

$$\textsc{let}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \leftarrow \text{Gen}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \to (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau_2}$$

$$\begin{array}{l} \text{Gen}(\tau, \Gamma) = \forall \vec{\alpha} . \tau \\ \text{where} \\ \vec{\alpha} = \text{FreeVars}(\tau) - \text{EnvFreeVars}(\Gamma) \\ \text{EnvFreeVars}(\Gamma) = \bigcup_{(x:\sigma) \in \Gamma} \text{FreeVars}(\sigma) \end{array}$$

Figure 14 – Let polymorphic λ-calculus

The LET rule introduces program variables with universal types. In an expression **let** $x = e_1$ **in** e_2, the type of x inside e_2 is a generalized version of the type of e_1, which is obtained by quantifying all the free type variables that do not already appear as free variables in types present in the environment Γ.

In the let-polymorphic type calculus, only variables can be polymorphically typed and the VAR rule is the place where these polymorphic types are used. If a variable x has an universal type σ according to the current type environment Γ, then uses of that variable can be assigned any type that is an instance of that universal type. These instances can be obtained by instantiating all the quantified variables in σ with suitable monotypes.

3.2.3 Principal Types and Type Inference

Some terms in the let-polymorphic λ-calculus can be assigned multiple types. For example, the identity function $\lambda x. x$ can be typed as either $\mathbf{bool} \to \mathbf{bool}$ or $\forall \alpha . \alpha \to \alpha$, as was previously mentioned.

These different types for a given term can be partially ordered based on how specific they are. Types with more quantifiers are more generic and types with less quantifiers are more specific.

One interesting property of the let-polymorphic λ-calculus is that each term has a single most general type, which is known as a *principal type*. This principal type can be found via an unification-based type inference algorithm [21; 22]. Finding principal types takes exponential time on some pathological inputs but is very efficient in practice [23; 24].

3.2.4 Static Guarantees

We initially presented universal types as a less restricted alternative to simple types. However, in this text we are framing type systems in terms of what they can statically guarantee about the behavior of the program when it is executed.

The additional guarantee that universal types offer over simple types is that if a function variable is universally quantified, then that variable will only be passed around to other functions or returned to the caller. It will not be used as the function in a function application expression and it will not be used as the conditional in an **if-then-else** expression. This can tell us a lot about the runtime behavior of a program. As shown by Wadler, each parametric type gives rise to a *free theorem* that is valid for all functions that have that type [25]. For example, in a purely functional language, the free theorem for the $\forall\alpha.\alpha \to \alpha$ type is that any function with that type is either the identity function or a function that enters an infinite loop and never returns.

3.3 Subtyping

Sometimes it is useful to organize types in hierarchies instead of as disjoint sets of values, as is done in the simply-typed λ-calculus. For a motivating example, consider the record types that we introduced in Section 2.7. Since record types list *all* the fields in the record, it is not possible for a function to receive inputs with extra fields that are not mentioned in the type of its domain. The following program executes successfully in an untyped setting but a simple type system for records cannot assign a type to the f function because each call site passes it an input of a different type.

$$\begin{array}{l}\textbf{let}\, f = \boldsymbol{\lambda} r.\, r.x \;\textbf{in} \\ (f\ \{x\!:\!10\}) + (f\ \{x\!:\!20,\ y\!:\!30\})\end{array}$$

3.3.1 The Subtyping Relation (<:)

Subtyping is characterized by a transitive and reflexive relation between types, the <: subtyping relation. A type S is said to be a subtype of a type T if any operation expecting a value of type T can also receive a value of type S. This is known as the *Liskov Substitution Principle* and can be formalized via the following *subsumption* rule:

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}$$

The precise definition of the <: subtyping relation depends on the type system and what features it contains. In the following subsections we describe how the subtyping relation works on records, variants and function types.

Subtyping Records

There are two ways that a record type can be a subtype of another. The first one is that a record type with more fields is a supertype of a record type with only a subset of those fields. This is known as *width-subtyping*. The second kind of subtyping is *depth–subtyping*. If two record types contain the same field names and the types of the contents of the fields in the first record type are all subtypes of the types of the respective fields in the second record type, then the first type is a subtype of the second.

$$\frac{}{\{\ell_1 : T_1, \ldots, \ell_n : T_n\} <: \{\ell_1 : T_1, \ldots, \ell_n : T_n, \ell_{n+1} : T_{n+1}\}}\ \textsc{record-width} \qquad \frac{S_i <: T_i}{\{\ell_i : S_i\} <: \{\ell_i : T_i\}}\ \textsc{record-depth}$$

Having a subtyping rule for record types is very useful in object-oriented programming languages, where class inheritance hierarchies map naturally to subtyping hierarchies.

Subtyping Variants

The subtyping rules for variants are very similar to the ones for records, except that width subtyping is the other way around. A type with more variants is a subtype of a type with only a subset of those variants.

$$\frac{}{<\ell_1 : T_1 \mid \ldots \mid \ell_n : T_n \mid \ell_{n+1} : T_{n+1}> \;<:\; <\ell_1 : T_1 \mid \ldots \mid \ell_n : T_n>}\ \textsc{variant-width}$$

$$\frac{S_i <: T_i}{<\ell_i : S_i> \;<:\; <\ell_i : T_i>}\ \textsc{variant-depth}$$

Subtyping Functions

The interesting thing about subtyping for function types is that the subtyping behaves differently in the domain and the codomain. A function that returns a more specific type can be used in the place of a function that returns a more general type. For the input type it is the other way around. A function that can receive a more general type can be used in the place of a function that receives a more specific type.

$$\frac{S' <: S \qquad T <: T'}{(S \to T) <: (S' \to T')}$$

3.3.2 The Top Type

Many type systems have the concept of a most general type that contains every single value, therefore being a supertype of every other type. In the type systems literature, this type is most often named Top. In object-oriented languages, this Top type is most often named Object, a name that is also borrowed by many non-OO languages.

$$\frac{}{S <: \text{Top}}$$

The only operations that can be performed on expressions of type Top are operations that are valid for every single type in the language. In the λ-calculi we described in this text there are no such operations, so expressions of type Top may only be passed around or returned. However, in some programming languages there are operations that work on any value. For example, some languages allow any value to be serialized into a string or to be tested for identity-based equality.

3.3.3 The Bottom Type

The dual of the Top type is the Bottom type, which is a subtype of every other type. The Bottom type is not inhabited by any values and can only be used on expressions that never produce a value when evaluated. One example of this are expressions that always fail by raising an exception.

$$\frac{}{\text{Bottom} <: S}$$

The presence of a Bottom type in the type system makes type inference more difficult. For example, in a language without bottom types the type of f in $(f\ x)$ must be a function type while in a language with the bottom type it can be either a function type or Bottom. Because of these extra difficulties, not every language with subtyping has a Bottom type. The Top type, on the other hand, is almost always present.

3.3.4 Variance

Variance is the name we use to describe the subtyping behavior a higher-order type constructor has depending on its parameter types. Type constructors can be either covariant, contravariant, or invariant.

Covariance	$C[S] <: C[T]$	iff	$S <: T$
Contravariance	$C[S] <: C[T]$	iff	$T <: S$
Invariance	$C[S] <: C[T]$	iff	$S = T$

For example, the function type constructor → is covariant in the type of the return values and contravariant in the type of the inputs. The most famous example of an invariant type constructor is the Ref$[T]$ type constructor for mutable reference cells. As a rule of thumb, locations out of which values flow (reads) are covariant, locations into where they flow (writes) are contravariant and locations where values flow both in and out (read and write) are invariant.

Additional evaluation rules $\boxed{e \longmapsto e'}$

$$\begin{array}{llll} C[\langle \tau \rangle\, v] & \longmapsto C[v] & v \in \tau & (\textsc{cast}) \\ C[\langle \tau \rangle\, v] & \longmapsto \textsc{cast-error} & v \notin \tau & (\textsc{cast-err}) \end{array}$$

Additional typing rules $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau <: \tau'}{\Gamma \vdash \langle \tau \rangle\, e : \tau}$$

Figure 15 – Runtime semantics and typing rule for the downcast operator

3.3.5 Downcasting

In languages with subtyping, the type of an expression can be freely converted to its supertype. Some languages provide a way to go in the other direction, and use values of a supertype as if they belonged to a subtype. This is known as *downcasting* and typically requires the programmer to add explicit type casts to their program, which we will denote by writing the name of the type in angle brackets: $\langle \tau \rangle\, e$.

Adding downcasts to a programming language has semantic consequences, which we describe in Fig. 15. When evaluated, a downcast can either succeed or raise a CAST-ERROR exception.

Poor-man's parametric polymorphism

The following program is an example of a program using downcasts. It upcasts an integer to the Top type and then downcasts it back to an integer so it can be used in an arithmetic operation.

$$\begin{array}{l} \textbf{let}\, x : \text{Top} = 1\ \textbf{in} \\ (\langle \textbf{int} \rangle\, x) + 2 \end{array}$$

One use of this style of downcasting is writing "generic" functions and data structures in programming languages that do not have parametric types. For example, the original collection classes in the standard library for the Java programming language work by storing values as Objects. When an user wants to insert an object into a collection, that object is first upcasted to the Object type. When the user reads values from the collection, he downcasts them back to the original type in order to be able to use them.

This style of programming is less type safe than using parametric polymorphism, since there is now a possibility of CAST-ERROR exceptions being raised at runtime. However, it is very appealing due to its simplicity. Downcasts are easy to add to a language with subtyping without adding a lot of complexity and cognitive overhead, which happens when adding parametric types to the type system. In languages with subtyping, parametric types need to have ways to specify the variance of the type parameters, which introduces significant complexity to an already complex type-system feature. Subtyping also pushes the language designer towards the more general *bounded-polymorphism*, which is even more complex than regular parametric polymorphism.

3.4 Union Types

The type system for the simply-typed λ-calculus partitions the set of values in distinct categories (ground types such as booleans and integers and various function types) and each expression is restricted to evaluate to values from only one of the categories. Union types make this more flexible, by making it possible to represent types that are the union of zero or more of those categories. A union type $\tau_1 \cup \tau_2$ represents the set of values that belong either to the τ_1 or to the τ_2 type.

Union types suggest a natural subtyping relation:

$$\textsc{union-sub}1 \quad \frac{}{S <: S \cup T} \qquad \textsc{union-sub}2 \quad \frac{}{T <: S \cup T}$$

The difference between unions and variant records is that unions are not tagged. The union **int** ∪ **int** is the same as the type **int**, which is not the case for the sum type **int** + **int**, where values tagged with the **left** tag are different from values tagged with the **right** tag.

The only operations that are allowed on expressions of a union type are operations that can work on any of the types in the union. One example of an operation that can work on multiple types is a tag-checking operation in a dynamically-typed language:

$$\begin{array}{llll} C[(\text{isInt } v)] & \longmapsto C[\text{true}] & v \in \textbf{int} & (\textsc{isInt-true}) \\ C[(\text{isInt } v)] & \longmapsto C[\text{false}] & v \notin \textbf{int} & (\textsc{isInt-false}) \end{array}$$

In dynamically-typed languages, values carry type information with them at runtime and programmers can use those tags directly instead of needing to create their own tags via variant records. For example, in a statically-typed language, a function to search for a value in a data structure may return a value of the Option variant type, but in a dynamic language it is more idiomatic to just return the result directly on success, without adding an additional wrapper to it, and null on failure.

```
// Idiomatic typed programming
type Option =<Just:int | Nothing:unit>
if (...) then (Just 10) else Nothing

// Idiomatic untyped programming
if (...) then 10 else null
```

Due to this programming idiom, many type systems for dynamic languages have some form of union types.

3.5 Intersection Types

Intersection types are the dual of union types. The intersection type $\tau_1 \cap \tau_2$ contains values that belong to both the τ_1 type *and* the τ_2 type.

$$\textsc{inter-sub}1\quad \frac{}{S \cap T <: S} \qquad\qquad \textsc{inter-sub}2\quad \frac{}{S \cap T <: T}$$

In the context of type systems for dynamic languages, the biggest use for intersection types is to allow *finitary overloading*, also known as *ad-hoc polymorphism*. For example, in many programming languages the addition operator $+$ is overloaded to work on either a pair of integers or a pair of floating point numbers. This can be typed in a system with function types by saying the type of the $+$ operation is an intersection of two function types.

$$+ : (\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \cap (\mathbf{float} \to \mathbf{float} \to \mathbf{float})$$

3.5.1 Intersection Types vs Union Types

Intersection types work for finitary overloading because the subtyping rules mean that the overloaded function can be seen as having either of the types in the intersection. For example, by applying the INTER-SUB1 rule to the type of the addition operator $+$, its type becomes $(\mathbf{int} \to \mathbf{int} \to \mathbf{int})$, which lets it be used on integers. Similarly, using the INTER-SUB2 rule lets it receive floating point inputs.

At first, it might seem that union types might be able to serve this same purpose. Since the $+$ operator can take inputs that are either **int**s or **float**s, one might expect that the following type would be a good fit for it:

$$((\mathbf{int} \cup \mathbf{float}) \to (\mathbf{int} \cup \mathbf{float}) \to (\mathbf{int} \cup \mathbf{float}))$$

However, this type is less precise than the alternative with intersection types because it does not enforce the correlations between the input and return types. It does not assert that the sum of two integers is also an integer and it does not enforce that both of the operands to the addition must have the same type.

4 Soft Typing

In dynamic languages some program operations may raise type errors at run time. For example, in the λ^{DYN} calculus from Section 2.3, function applications may raise a NOT-A-FUNCTION exception if they receive null as their input. Soft typing is a family of static analysis techniques that use type inferencing to classify the uses of these potentially error-raising operations in a dynamically-typed program into three cases:

Provably safe operations always receive well-typed inputs

Potentially unsafe operations may or may not receive well-typed inputs.

Provably unsafe operations never receive well-typed inputs.

Provably safe operations present opportunities for optimization, from being able to skip runtime tag checks or from allowing the use of untagged memory representations for some values [26].

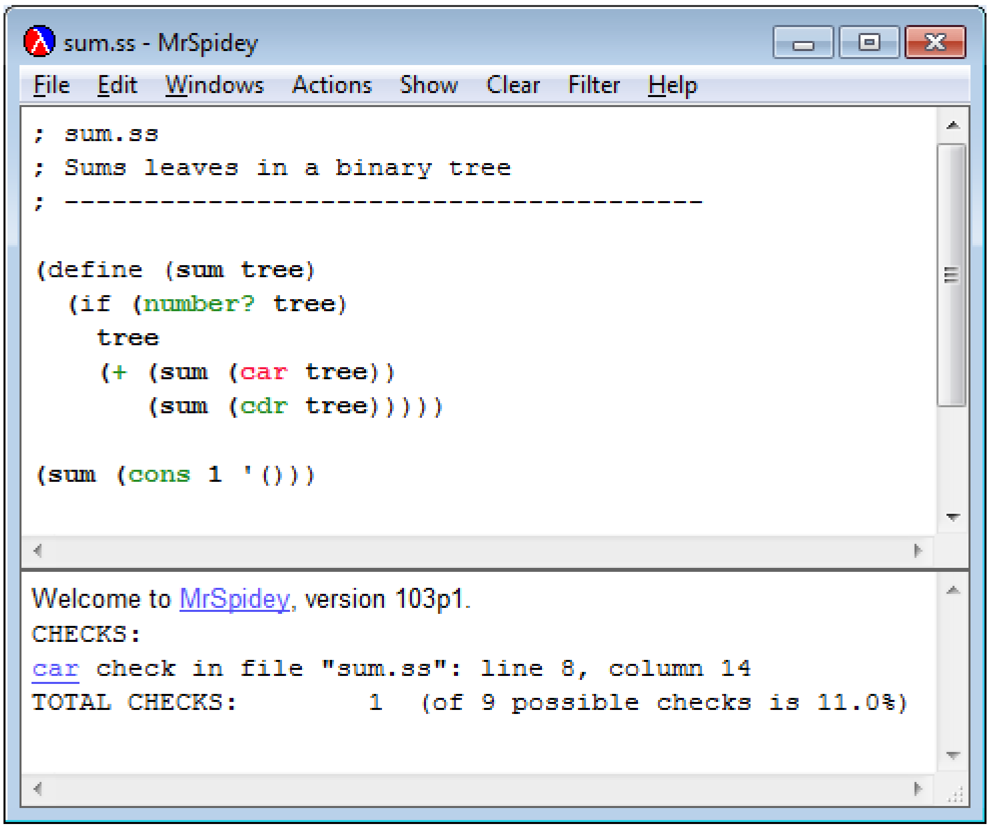
Provably unsafe operations most likely correspond to program bugs because they will always raise an error if they are evaluated. According to Sagonas [27], these provably unsafe operations are most often found in error-handlers and other rarely executed code paths.

Potentially unsafe operations happen in parts of the program that the type system could not reason about precisely. This is an inevitable consequence of the presence of runtime type errors being a nontrivial program property that is undecidable according to Rice's Theorem [15].

Fig. 16 contains an example of this classification in the graphical interface of the MrSpidey soft typing system [28]. MrSpidey colors in green all the provably safe operations in the program and underlines and colors in red all the remaining operations. For example, the first **car** operation is marked in red because according to MrSpidey's reasoning, that operation could possibly receive a **nil** value as input (which is in fact what happens if we run the program). Finally, MrSpidey presents a report summarizing what percentage of the operations in the program could not be proven to be safe (1 out of 10). MrSpidey does not differentiate between potentially unsafe and provably unsafe operations but this is not due to a limitation of its algorithm. We will further discuss theses error reporting choices in Section 4.4.

Note that MrSpidey, like all other soft type checkers, does not check the types of function arguments or return values. All warnings point to value destruction sites such as function applications and arithmetic operations because these are the program locations that might actually raise exceptions at runtime.

Another thing that can be noted is the lack of type annotations in the program in Fig. 16. The original goal behind the development of soft typing systems was to add a type verification step to scripting languages being used for rapid prototyping, without interfering with their flexibily by requiring additional type annotations [29]. Because of this requirement, type annotations would not make much sense in a soft type system setting. Since soft type



```
; sum.ss
; Sums leaves in a binary tree
; ----------------------------------------

(define (sum tree)
  (if (number? tree)
    tree
    (+ (sum (car tree))
       (sum (cdr tree)))))

(sum (cons 1 '()))
```

```
Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "sum.ss": line 8, column 14
TOTAL CHECKS:        1  (of 9 possible checks is 11.0%)
```

Figure 16 – MrSpidey's program analysis result window

systems do not change the semantics of their underlying dynamically-typed language the type annotations cannot have any semantic meaning and since the type inference algorithms must be able to infer types for completely unannotated programs the type system must not have any features that require type annotations for inference.

In the remainder of this chapter, we will present two soft type systems for a dynamic λ-calculus with booleans as the only ground type. The first soft type system is based on the type system of Cartwright and Fagan [30; 31], which uses ML-like type inference via unification and the second one is based on the safety analysis of Palsberg and O'Keefe [32], which uses flow analysis for inference. Most of the soft typing systems in the literature follow one of these two approaches.

4.1 An Unification-based Soft Type System

In this section we present a soft type system based on the type systems of Fagan, Wright and Cartwright. Fagan and Cartwright introduced the first soft type system and also coined the term "Soft Typing" [30; 31]. Wright and Cartwright later expanded Fagan's system to cover the full Scheme language, including features such as assignment, variable-arity functions and first-class continuations [33; 34]. Our simplified type system is closer to Fagan's original system but some of our presentation is closer to Wright's.

We divide our exposition of the system into two parts. Firstly we describe a static type system and corresponding inference algorithm that can reason about common patterns found in dynamically-typed programs. Secondly we describe how to soften this type system, making it classify operations according to their safety instead of simply rejecting programs that cannot be proven to be totally safe.

4.1.1 Static Types in Fagan's Type System

To determine if an operation is provably safe or not in a λ-calculus with booleans as the only ground type there are 4 cases that matter for the inputs of that operation:

1. Expressions that always evaluate to booleans
2. Expressions that always evaluate to a function
3. Expressions that might evaluate to either a boolean or a function
4. Expressions that never evaluate to any value (for example, expressions that abort the execution of the program)

The central part of Fagan's type system is the use of union types and recursive types to model these four cases. As shown in Fig. 17, monotypes in this type system are the union of a boolean part and a function part. The boolean part being $\mathbf{bool}^{+}$, indicates that expressions with that type may evaluate to a boolean. Conversely, the boolean part being $\mathbf{bool}^{-}$, indicates that expressions with that type never evaluate to a boolean. The function part behaves similarly. The function part being $(\tau_1 \rightarrow^{+} \tau_2)$ means that the type contains functions which receive τ_1 and return τ_2 and if the function part is $\rightarrow^{-}$ then expressions with that type never evaluate to a function.

In addition to the union types, Fagan's type system also features parametric polymorphism. Recall that the instantiation relation $\tau \sqsubseteq \sigma$ means that there is an instantiation of the quantified variables in σ that results in τ and that the generalization function $Gen(\tau, \Gamma)$ returns a parametrically polymorphic type created by universally quantifying all the free type variables in τ that are not bound in Γ.

Fig. 18 provides some concrete examples of these union types. $\mathbf{bool}^{+} \cup \rightarrow^{-}$ is the boolean type, for values that may be either true or false and may not be functions; $\mathbf{bool}^{-} \cup \rightarrow^{-}$ is the empty type, for expressions that never return a value; $\mu\alpha.\mathbf{bool}^{+} \cup (\alpha \rightarrow^{+} \alpha)$ is the recursive type for expressions that may evaluate to any value.

The most notable feature of Fagan's static type system is that it has union types but does *not* have any form of subtyping. The reason for this is that type inference efficiency is of paramount important in a soft type system (since programs have no type annotations) and, according to Wright, type inference for type systems with structural subtyping and recursive types "consumes exorbitant amounts of memory and execution time for even small examples" [33, p135].

Instead of subtyping, Fagan's type system provides the required flexibility for unions via type variables. This technique is often called *row polymorphism* [22; 35] and, according to Fagan, it was inspired on Remy's record

Type Syntax

Type Variable	α	$::=$	$\alpha, \beta, \ldots$
Monotype	τ	$::=$	$(\varphi_B \;\cup\; \varphi_F) \mid \alpha \mid \mu\alpha.\tau$
Boolean Part	φ_B	$::=$	$\mathbf{bool}^- \mid \mathbf{bool}^+$
Function Part	φ_F	$::=$	$\rightarrow^- \mid (\tau_1 \rightarrow^+ \tau_2)$
Polytype	σ	$::=$	$\tau \mid \forall\alpha.\sigma$

Typing rules $\boxed{\Gamma \vdash e : \tau}$

VAR
$$\frac{\Gamma(x) = \sigma \qquad \tau \sqsubseteq \sigma}{\Gamma \vdash x : \tau}$$

BOOL
$$\frac{b \in \text{Boolean} \qquad \tau \sqsubseteq \forall\varphi_F(\mathbf{bool}^+ \cup \varphi_F)}{\Gamma \vdash b : \tau}$$

IF
$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq \forall\varphi_B(\varphi_B \cup \rightarrow^-) \\ \Gamma \vdash e_2 : \tau_2 & \Gamma \vdash e_3 : \tau_2\end{array}}{\Gamma \vdash \mathbf{if}\; e_1 \;\mathbf{then}\; e_2 \;\mathbf{else}\; e_3 : \tau_2}$$

LAM
$$\frac{\Gamma[x \leftarrow \tau_1] \vdash e : \tau_2 \qquad \tau \sqsubseteq \forall\varphi_B.(\varphi_B \cup (\tau_1 \rightarrow^+ \tau_2))}{\Gamma \vdash (\boldsymbol{\lambda} x.\, e) : \tau}$$

APP
$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq (\mathbf{bool}^- \cup (\tau_2 \rightarrow^+ \tau_3)) \\ \Gamma \vdash e_2 : \tau_2 & \end{array}}{\Gamma \vdash (e_1\; e_2) : \tau_3}$$

APP-EMPTY
$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq (\mathbf{bool}^- \cup \rightarrow^-) \\ \Gamma \vdash e_2 : \tau_2 & \end{array}}{\Gamma \vdash (e_1\; e_2) : \tau_3}$$

LET
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \leftarrow Gen(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \rightarrow (\mathbf{let}\, x = e_1 \;\mathbf{in}\, e_2) : \tau_2}$$

$$\text{Gen}(\tau, \Gamma) = \forall\vec{\alpha}.\tau$$
where
$$\vec{\alpha} = \text{FreeVars}(\tau) - \text{EnvFreeVars}(\Gamma)$$
$$\text{EnvFreeVars}(\Gamma) = \bigcup_{(x:\sigma)\in\Gamma} \text{FreeVars}(\sigma)$$

Figure 17 – Fagan's (non-soft) type system

Nothing	$\mathbf{bool}^- \cup \rightarrow^-$
Boolean	$\mathbf{bool}^+ \cup \rightarrow^-$
"Bool to Bool" function	$\mathbf{bool}^- \cup ((\mathbf{bool}^+ \cup \rightarrow^-) \rightarrow^+ (\mathbf{bool}^+ \cup \rightarrow^-))$
Anything	$\mu\alpha.(\mathbf{bool}^+ \cup (\alpha \rightarrow^+ \alpha))$

Figure 18 – Examples of types in Fagan's type system

system for ML [36]. The basic idea behind row polymorphism is that instead of "throwing away" information from the more specific type when converting it to a more general type, a *row variable* φ is used to represent the difference between the more specific and the more general type. Any type inequality constraints in the program get converted into equality constraints with a row variable acting as a slack variable, as is illustrated in the following equations:

$$A \subseteq B$$
$$A \cup \varphi = B$$

These equality constraints with slack variables are not as powerful as real subsumption (as will be discussed in Section 4.3) but they allow for an efficient unification-based type inference algorithm while still providing lots of flexibility to the type system.

For a concrete example of this use of slack variables, consider the typing rules BOOL and LAM for boolean and function literals. If boolean literals had type $(\mathbf{bool}^+ \cup \rightarrow^-)$ and functions had type $(\mathbf{bool}^- \cup (\tau_1 \rightarrow^+ \tau_2))$, without any slack variables, it would not be possible to type the following program:

$$\lambda b.\, \mathbf{if}\ b\ \mathbf{then}\ \text{true}\ \mathbf{else}\ (\lambda x.\, x)$$

Without slack variables, the types of the expressions in the **then** and **else** branches would be different but the IF rule requires them to be the same. With slack variables, both branches can be given the type $(\mathbf{bool}^+ \cup (\alpha \rightarrow^+ \alpha))$.

The other unintuitive aspect of the typing rules is how the IF and APP rules use polymorphic types for their input parameters instead of just saying that they receive exactly booleans and functions, respectively. The reason for this is to allow the input expression to have the empty type. This mitigates the *reverse flow* problem, which is mentioned in Section 4.3.

4.1.2 Softening Fagan's Type system

The presence of union types allows the type system that we have presented so far to reason about code that mixes booleans and functions. However it is not yet a complete soft type system because it either determines that all operations are provably safe or it gives up without pinpointing which operations are the potentially unsafe ones.

The following two programs are examples that cannot be typed in the non-soft system we have presented so far. Program P fails to typecheck because of the unsafe function application. Program Q evaluates successfully and is only untypable because the type system is not sufficiently expressive.

$$P = (\text{true false})$$
$$Q = \lambda b.\,\mathbf{if}\ b\ \mathbf{then}\ b\ \mathbf{else}\ (\lambda x.\,x)$$

The reason these programs fail to typecheck is the presence of conflicting positive and negative type information in the typing rules. The P program cannot be typed because the only typing rules for applications, APP and APP-EMPTY, require that the function parameter does not have a boolean part. As for the Q program, the IF rule demands that both the **then** and **else** branches have the same type, meaning that the type of the b in the **then** branch must include the type of the functions from the **else** branch. However, b is also used as the conditional in the **if**-expression and the IF rule requires that the type of the conditional does not contain a function part.

Since all type-inference failures come from conflicts between rules that require the presence of a type and rules that require its absence, one way to soften the type system is to get rid of all the rules requiring that a type be absent. As shown in Fig. 19 the changes all involve replacing instances of negative type information in the typing rules (ie. $\mathbf{bool}^-$ and $\rightarrow^-$) by a *soft type variable* $\tilde{\varphi}$. These soft type variables act as slack variables during type inference and behave similarly to regular universally-quantified variables. The difference between soft type variables and regular slack variables is that once type inference is complete the soft type checker checks how the soft type variables were instantiated to determine the safety of each program operation:

- In provably safe operations the soft type variables can be instantiated to an empty type, such as $\mathbf{bool}^-$ or $\rightarrow^-$.
- In potentially unsafe operations both the soft and non-soft variables are instantiated to non-empty types such as $\mathbf{bool}^+$ or $\alpha \rightarrow^+ \beta$.
- In provably unsafe operations the soft type variables are instantiated to a non-empty type and the associated non-soft type variables are instantiated to an empty type.

4.1.3 A Concrete Example

Lets return to the program Q that we previously mentioned. Under the more flexible soft typing rules, the variable b can now be given the type $(\mathbf{bool}^+ \cup (\alpha \rightarrow^+ \alpha))$. The $\mathbf{bool}^+$ part is required because the function is called with true as a parameter and the $(\alpha \rightarrow^+ \alpha)$ part is because both branches of the if statement must have the same types. The softness comes into play in the IF-SOFT rule, where the $\tilde{\varphi}_F$ variable gets instantiated to $(\alpha \rightarrow^+ \alpha)$. Since the soft type variable was instantiated to a non-empty type, the type checker produces a warning saying that the condition in the **if-then-else** expression is potentially unsafe.

$$\textsc{if-soft}\quad \frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq \varphi_B \tilde{\varphi}_F(\varphi_B \cup \tilde{\varphi}_F) \\ \Gamma \vdash e_2 : \tau_2 & \Gamma \vdash e_3 : \tau_2\end{array}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau_2}$$

$$\textsc{app-soft}\quad \frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq \forall \tilde{\varphi}_B(\tilde{\varphi}_B \cup \tau_2 \rightarrow^{+} \tau_3) \\ \Gamma \vdash e_2 : \tau_2 & \end{array}}{\Gamma \vdash (e_1\ e_2) : \tau_3}$$

$$\textsc{app-empty-soft}\quad \frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau_1 & \tau_1 \sqsubseteq \forall \tilde{\varphi}_B(\tilde{\varphi}_B \cup \rightarrow^{-}) \\ \Gamma \vdash e_2 : \tau_2 & \end{array}}{\Gamma \vdash (e_1\ e_2) : \tau_3}$$

Figure 19 – Soft typing rules for Fagan's type system

4.2 A Flow-based Analysis Soft Type System

In this section, we present a soft type system based on the *safety analysis* of Palsberg and Schwartzbach [37], a global static analysis technique similar to Shiver's 0CFA, a context-insensitive form of *control-flow analysis* [38; 39].

The aim of safety analysis is to compute an approximation for what values flow to each expression and function parameter in the program and use this to determine if runtime type errors may occur. If the flow analysis can prove that only booleans flow into a certain variable, then its safe to use that variable in a conditional. On the other hand, if the analysis concludes that both booleans and functions might flow into a variable then there is a possibility of a runtime error if that variable is used in a conditional. The hard part of safety analysis is that in programming languages with higher order functions it is impossible to statically compute the control-flow graph of the program. The solution taken in 0CFA and in Palsberg's safety analysis is to approximate the control-flow graph by tracking to where each lambda in the program might flow as part of the value-flow computation.

Safety analysis partitions the set of values in the core language into disjoint sets. As shown in the following table, booleans are grouped together in the **bool** set and function abstractions are grouped together according to which λ-expression in the original program they originate from. We could have labeled each λ-abstraction in the original program with an unique label but, since in a call-by value semantics the variable name of a function abstraction never needs to be renamed, we instead refer to functions by their variable, to keep the notation lighter. Another way to look at this approximation of the set of function values is that in an environment-based semantics, with function closures instead of substitution, our approximation is equivalent to

Phrase:	Basic constraints:
b	$\{\mathbf{bool}\} \subseteq [\![b]\!]_P$
$\boldsymbol{\lambda} x.\, e$	$\{\boldsymbol{\lambda} x\} \subseteq [\![\boldsymbol{\lambda} x.\, e]\!]_P$

Phrase:	Connecting constraints:
$(e_1\ e_2)$	For every $\boldsymbol{\lambda} x.\, e_3$, if $\boldsymbol{\lambda} x \in [\![e_1]\!]_P$ then $[\![e_2]\!]_P \subseteq [\![x]\!]_P$ $[\![e_3]\!]_P \subseteq [\![(e_1\ e_2)]\!]_P$
$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	$[\![e_2]\!]_P \subseteq [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]_P$ $[\![e_3]\!]_P \subseteq [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]_P$

Phrase:	Safety constraints:
$(e_1\ e_2)$	$[\![e_1]\!]_P \subseteq \{\boldsymbol{\lambda}\_\}$
$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	$[\![e_1]\!] \subseteq \{\mathbf{bool}\}$

Figure 20 – Set constraints for safety analysis via 0CFA

ignoring the captured environment of closure values.

$$\begin{aligned} \text{Concrete Value } v &::= \text{true} \mid \text{false} \mid \boldsymbol{\lambda} x.e \\ \text{Abstract Value } \hat{v} &::= \mathbf{bool} \mid \boldsymbol{\lambda} x \end{aligned}$$

The *abstract-value* name comes from the *abstract interpretation* literature. 0CFA can be seen as a form of abstract interpretation and the abstract interpretation framework leads to many more powerful alternatives to 0CFA. However in this text we will keep to a simpler presentation based on systems of equations.

For a given program P, safety analysis computes an approximation $[\![\ ]\!]_P$: $(\text{Expr} \cup \text{Var}) \to 2^{\hat{v}}$ that maps program subexpressions and variable bindings into a set of abstract values. This approximation is specified by a set of set-inclusion restrictions. The reason the approximation depends on P is that safety analysis is a form of *whole-program* analysis. The approximations for the function values depend on what are the λ-abstractions in the original source code of P.

Each subexpression in P produces one or more set-inclusion restrictions, which can be read as a series of "flows-to" relations between subexpressions in the program. The full list of rules is shown in Fig. 20. For each boolean literal subexpression b in P, a $\{\mathbf{bool}\} \subseteq [\![b]\!]_P$ constraint is generated, for each function literal subexpression $\boldsymbol{\lambda} x.\, e$, a $\{\boldsymbol{\lambda} x\} \subseteq [\![\boldsymbol{\lambda} x.\, e]\!]_P$ constraint is generated and so on.

If a solution exists for the system of equations that safely analysis produces for P then the *safety constraints* will be satisfied, which allows us to say that the program is provably safe. If there are no solutions then the program is deemed potentially unsafe.

Proof sketch for the correctness of Safety Analysis

To prove that the system of equations is a sound approximation of the evaluation of the program we must show that if a subexpression e may evaluate to a value v during the evaluation of the program, then we must have $\hat{v} \in [\![e]\!]_P$. That is, if e evaluates to a boolean then we must have **bool** $\in [\![e]\!]_P$ and if e evaluates to a function $\lambda x.e$ then we must have $\lambda x \in [\![e]\!]_P$. For variables we also need to prove a similar property: if a variable x may be bound to a value v at some point of the program execution, then $\hat{v}$, the abstraction of that value, must be a member of $[\![x]\!]_P$.

The evaluation of the program can be represented by a derivation tree built using the big-step semantics of the language we are using. The proof of soundness analysis can be found via an induction on these evaluation trees.

4.2.1 Softening the Safety Analysis

Similar to what happened in our exposition of Fagan's type system, our initial version of safety analysis is not suitable as a soft type checker because it does not pinpoint which operations are the potentially unsafe ones, if any such operations exist. One way to soften the type system is to ignore the safety constraints when solving the system of equations, using only the basic constraints and connecting constraints for that. Without the safety constraints, there is always at least one solution to the system, the *supremum solution* that assumes that all values may flow to all program locations.

What the soft type checker then does is compute the least solution for the system of inequalities without the safety constraints (considering a partial ordering based on set inclusion) and check if that least solution also happens to satisfy the safety constraints. Each satisfied safety constraint corresponds to a provably safe operation and each unsatisfied safety constraint corresponds to an unsafe operation. For an unsatisfied constraint $A \not\subseteq B$, if $A \cap B \neq \emptyset$ then the operation is potentially unsafe. If $A \cap B = \emptyset$ then the operation is provably unsafe.

4.2.2 Finding Solutions For the System of Constraints

It is possible to find a minimal solution for the system of equations from 0CFA in $O(n^3)$ time via an iterative process that starts by assigning the empty set to every variable and proceeds by relaxing constraints until a fixed point is reached [38; 40; 41]. However, for our purposes it suffices to show that a least solution exists and that an algorithm can find it, no matter how inneficiently. Since the intersection of two solutions is also a solution, the intersection of all solutions will be the least solution. This least solution must exist because the set of solutions is non-empty (the supremum solution is always there). Finally, since the search space for the constraint system is finite (the number of variables is finite, as is the number of primitive types and lambda abstractions) it is possible to enumerate all the solutions to the system by brute force, which will end up finding the least solution.

4.2.3 A Concrete Example of 0CFA

To demonstrate 0CFA in action, we ideally want a program using higher order functions. One of the simplest ones that fits that description is the following:

$$P = ((\lambda id.\, ((id\ id)\ \text{true}))\ (\lambda x.\, x))$$

The 0CFA constraint system and the least solution for this program are shown in Fig. 21 and Fig. 22. To find out if the soft type system emits warnings for this program we check the operations that may potentially raise runtime errors, which are the three function applications: $(id\ id)$, $((id\ id)\ 1)$ and $((\lambda id.\, (...))(\lambda x.\, x))$.

1. The first function application, $(id\ id)$, is provably safe because $[\![id]\!]_P = \{\lambda x\}$ only contains functions.

2. The second function application, $((id\ id)\ 1)$, is potentially unsafe because the flow analysis concludes that the *id* function could return either functions or booleans so $(id\ id)$ is not guaranteed to be a function.

3. The third function application is provably safe because $[\![(\lambda id.(...))]\!]_P = \{\lambda id\}$ only contains functions.

In this particular example, the program evaluates without errors and the warning from the potentially unsafe operation is a false positive caused by 0CFA inferring monomorphic types for all identifiers.

4.2.4 Interpreting 0CFA as a Type System

0CFA computes set approximations that range over labeled function expressions. This is not very similar to typical type systems, where function types are described by the types of their domains and codomains. While keeping track of the function labels is necessary for approximating the program control flow graph and is also helpful for providing helpful error messages, this level of detail is not needed if we only want to report which operations are provably safe and which are not. For this, it suffices to determine what expressions may evaluate to booleans, what expressions may evaluate to functions (any function at all), and what expressions may evaluate to some combination of both. Palsberg and O'Keefe show [32] that the simply-typed lambda calculus with recursive types and subtyping [42] is equivalent to 0CFA, in that it will statically type the same programs that 0CFA accepts without any warnings.

However, this will be useful mostly just as a comparison to other soft type systems. There are no unification-based type inference algorithms for the simply-typed lambda calculus with recursive types and subtyping [43] so the only way to infer the types is to compute the 0CFA approximation and then translate it to the type-system formulation (or use another equivalent algorithm).

Type T ::= $\mathbf{bool} \mid T_1 \to T_2 \mid \alpha \mid \mu\alpha.T \mid \bot \mid \top$

$$\mathbf{bool} \in [\![1]\!]_P \qquad \lambda x \in [\![(\lambda x.x)]\!]_P \qquad \lambda id \in [\![(\lambda id.(...))]\!]_P$$

$$\begin{array}{ll}
\text{If } \lambda x \in [\![id]\!]_P \text{ then} & \text{If } \lambda id \in [\![id]\!]_P \text{ then} \\
\quad [\![id]\!]_P \subseteq [\![x]\!]_P & \quad [\![id]\!]_P \subseteq [\![id]\!]_P \\
\quad [\![x]\!]_P \subseteq [\![(id\ id)]\!]_P & \quad [\![((id\ id)\ 1)]\!]_P \subseteq [\![(id\ id)]\!]_P
\end{array}$$

$$\begin{array}{ll}
\text{If } \lambda x \in [\![(id\ id)]\!]_P \text{ then} & \text{If } \lambda id \in [\![(id\ id)]\!]_P \text{ then} \\
\quad [\![1]\!]_P \subseteq [\![x]\!]_P & \quad [\![1]\!]_P \subseteq [\![id]\!]_P \\
\quad [\![x]\!]_P \subseteq [\![((id\ id)\ 1)]\!]_P & \quad [\![((id\ id)\ 1)]\!]_P \subseteq [\![((id\ id)\ 1)]\!]_P
\end{array}$$

$$\begin{array}{ll}
\text{If } \lambda x \in [\![(\lambda id.(...))]\!]_P \text{ then} & \text{If } \lambda id \in [\![(\lambda id.(...))]\!]_P \text{ then} \\
\quad [\![(\lambda x.x)]\!]_P \subseteq [\![x]\!]_P & \quad [\![(\lambda x.x)]\!]_P \subseteq [\![id]\!]_P \\
\quad [\![x]\!]_P \subseteq [\![((\lambda id.(\ldots))(\lambda x.x))]\!]_P & \quad [\![((id\ id)\ 1)]\!]_P \subseteq [\![((\lambda id.(\ldots))(\lambda x.x))]\!]_P
\end{array}$$

Figure 21 – Constraint system for the example program P

$$\begin{array}{rcl}
[\![1]\!]_P & = & \{\mathbf{bool}\} \\
[\![x]\!]_P & = & \{\mathbf{bool}, \lambda x\} \\
[\![id]\!]_P & = & \{\lambda x\} \\
[\![(\lambda x.x)]\!]_P & = & \{\lambda x\} \\
[\![(id\ id)]\!]_P & = & \{\mathbf{bool}, \lambda x\} \\
[\![((id\ id)\ 1)]\!]_P & = & \{\mathbf{bool}, \lambda x\} \\
[\![(\lambda id.(...))]\!]_P & = & \{\lambda id\} \\
[\![((\lambda id.(\ldots))(\lambda x.x))]\!]_P & = & \{\mathbf{bool}, \lambda x\}
\end{array}$$

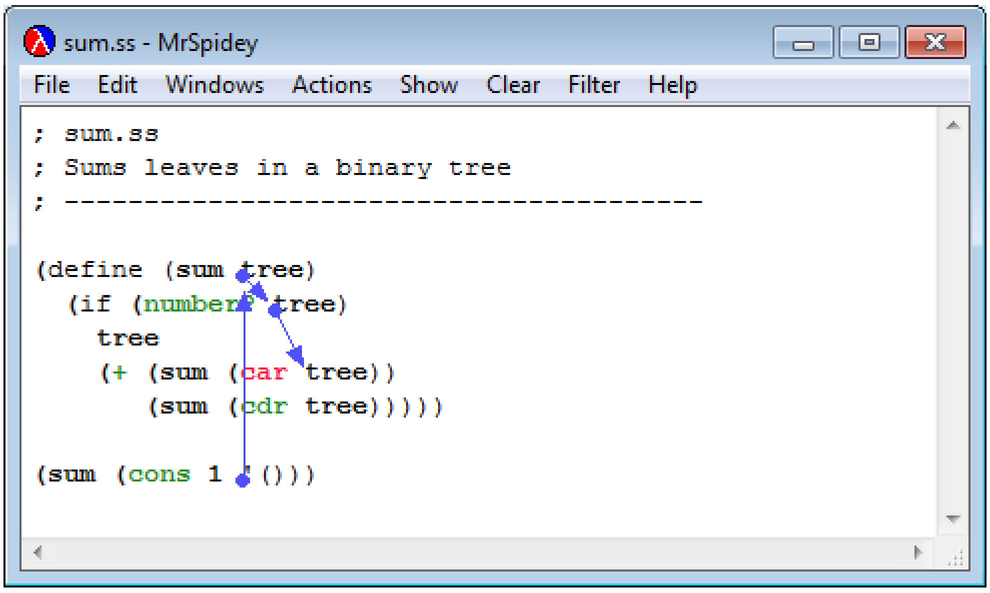
Figure 22 – Result of 0CFA for the example program P

4.3 Unification-based vs Flow-based Soft Typing

There are two main advantages for unification-based algorithms. The first one has to do with performance. While unification-based type inference has exponential worst case runtime in the presence of parametric polymorphism [23], in practice unification-based inference is has almost linear runtime [22; 24]. Another positive feature of unification-based algorithms is that they infer the most general type for each expression, which allows for an easy implementation of parametric polymorphism and separate compilation of modules.

Inference algorithms based on control flow analysis take more time to run (worst case cubic time and often at least quadratic time), specially once polymorphism is added (some variations of flow-based-analysis, such as 1CFA, have exponential time complexity). Control flow analysis is also a form of global program analysis and therefore does not easily lend itself to separate compilation. Flanagan had to introduce contraint-simplification heuristics to make it feasible to run his flow-based inference algorithm on programs with many modules [44].

That said, an advantage of control-flow analysis is that it can be more accurate than unification-based analysis. Type information only flows in a forward direction, unlike what happens in unification algorithms, where the



```
; sum.ss
; Sums leaves in a binary tree
; ----------------------------------------

(define (sum tree)
  (if (number? tree)
     tree
     (+ (sum (car tree))
        (sum (cdr tree)))))

(sum (cons 1 '()))
```

Figure 23 – Flow-based inference can point out the origins of the values that may cause potential runtime exceptions

ubiquitous presence of equality constraints means that type information can flow both ways.

This additional accuracy also allow control-flow analysis to give more understandable error messages. While error messages in unification-based inference algorithms only mention that the inferred types for two things do not match (which does not always point towards the real cause of the problem), flow-based analysis can often pinpoint the precise reason for a type mismatch. For example, the GUI for the MrSpidey typechecker can display an "inference-trace" that graphically displays what values flow into a given expression, as shown in Fig. 23. Another example not related to soft typing is the Helium compiler for Haskell [45]. Helium is focused on educational applications and uses a flow-based type inference algorithm to generate error messages that are more easily understandable than the messages typically generated by Haskell compilers. Heeren's thesis on Helium includes a good explanation on why flow-based type inference can lead to better error messages [46].

4.4 Success Types

According to Felleisen, who during his time at Rice university was part of many research groups working on soft-typing systems for Scheme, the soft-typing experiment was a failure [29]. According to him, the soft typing systems for Scheme had a tendency to infer very large and complex types, which resulted in error messages that were very difficult to comprehend. It was also hard to know just from reading the error message, what fix should be applied to the program. Programmers would often ignore the error messages from the soft-typing engine and fall back to using a traditional debugger to step through the buggy program.

However, there is one soft-typing system that has managed to achieve some form of success outside academia. Dialyzer, the DIscrepancy AnalYZer for ERlang, is a static analyzer for the Erlang programming language based on soft-typing principles [47]. Dialyzer is very popular among Erlang programmers and, according to a 2008 survey, it was the most used tool for testing or statically-analyzing Erlang programs [48]. Dialyzer is currently bundled with the standard Erlang/OTP distribution [49] and is also covered in introductory books for Erlang [50].

What did Dialyzer do differently from other soft typing systems to achieve this status? The distinctive feature of Dialyzer is that, unlike other soft-typing systems, it does *not* create warnings for potentially unsafe operations. It only outputs warning messages for operations that are provably unsafe. One of the main reasons for this is to avoid overwhelming the programmer with a large number of error messages for potentially unsafe operations that are actually perfectly safe. According to the Dialyzer developers, if there are too many of these "false alarms", programmers will avoid using the static analysis tool [51].

To justify the unusual error-reporting strategy behind Dialyzer, Lindahl and Sagonas introduced the concept of *success typings* [47]. The purpose of traditional type systems is to rule out all programs that go wrong. On the other hand, the purpose of success types is to rule in all programs that go right.

Definition (Success Typings)**.** A success typing of a function f is a type signature $\alpha \to \beta$, such that whenever an application $(f\ x)$ reduces to a value v, then v is a β and x is an α.

In a traditional type system if a function f has the type $\alpha \to \beta$, it means that if given an α as input, it will certainly return a β. On the other hand, in a success-typing framework if a function f has the type $\alpha \to \beta$, it means that it is guaranteed to go wrong if it receives an input that is not an α and that it never returns a value that is not a β.

If we look at types as sets of values, the traditional type of an expression is a conservative subset of the values that expression may evaluate to, with the goal of ensuring that no well-typed programs go wrong. Success types are the other way around. The success type of an expression is a conservative superset of the values that expression may evaluate to, with the goal of ensuring that all programs that go right are well-typed.

One consequence of the success-typing approach is that in a type system based on success types, every function can be assigned the type **any** → **any**, where **any** is the Top type in Dializer's type system. While giving a function this permissive type will never detect an unsafe application of it, it allows Dialyzer to analyze a module even if the definitions of some of the functions it is using are unknown. Meanwhile, the equivalent trivial function type in a traditional type system is $\bot \to \top$, which is not very useful because a function with this type cannot be applied to any inputs.

Success-typing also offers a solution to the problem of large types resulting in complex and incomprehensible error messages. If during the type inference process the type of that term gets too big and complicated, Dialyzer relaxes the type of that term to **any** [1] . This choice of a less precise type causes more operations to be classified as potentially unsafe. However, in a success-typing framework this is not as big of problem as it is for traditional type systems. In a traditional type system, increasing the set of potentially unsafe programs increases the chance that a perfectly functioning program will be rejected by the type system, which forces the programmer to adapt the program just to appease the type systems. To avoid this, type system designers have a strong incentive to make their types as powerful and flexible as they can. On the other hand, in a success-typing framework making types less precise will just cause some provably unsafe operations to stop being detected. The type system designer has more freedom to make tradeofs between the accuracy of the type system and its simplicity and runtime efficiency.

[1] Another reason why Dialyzer relaxes some inferred types to **any** is to ensure that its inference algorithm terminates, by preventing inferred types from growing infinitely large.

5 Type Specifiers in Common LISP

In this chapter, we discuss the type declaration system in Common LISP [52], which was one of the first dynamic languages with a system for optional type annotations. Differently from the more recent uses of types for dynamic languages, in Common LISP the type annotation system was primarily meant as an aid for program optimization, since at the time computers had very limited processing power and program efficiency was a very important concern.

We first describe how Common LISP uses type declarations to guide compiler optimizations and then discuss the effect that they have on the language semantics and how Common LISP sacrifices soundness in the name of these optimizations.

5.1 Specialized operations in Common LISP

One approach dynamic languages can take to allow programmers to extract extra performance in hot code paths is to provide type-specialized operations that skip runtime checks. For example, in MACLISP the `+$`, `-$`, `*$` and `/$` operators are specialized for floating-point numbers [53]. In the following MACLISP program, which computes the discriminant of the quadratic formula, these operators will be compiled down to efficient floating-point arithmetic instructions without any runtime tag checks. It is the responsibility of the programmer to ensure that the `discriminant` function only receives floating-point inputs because the behavior of the type-specialized operators is undefined if they receive non-floating-point inputs.

```
(defun discriminant (a b c)
  (-$ (*$ b b) (*$ 4.0 a c)))
```

The Common LISP language standard descended from MACLISP and also supports type-specialized operations [52]. However, instead of providing specialized operations for every possible type and combination of types (single-precision and double-precision floats, fixed-width integers, rational numbers, etc.) in Common LISP the only arithmetic operators are the generic +,−,* and / operators. Specialization occurs through type-directed optimizations guided by type declarations. In the following program, the `a`, `b` and `c` parameters are all declared to be single-precision floating-point numbers and an optimizing Common LISP implementation is allowed to replace the generic arithmetic operations with floating-point ones, just like what was done by hand in the previous MACLISP example.

```
(defun discriminant (a b c)
  (declare (single-float a b c))
  (- (* b b) (* 4.0 a c)))
```

In addition to optimizing arithmetic expressions, type annotations in Common LISP can also optimize a variety of other operations and data structures. The following program illustrates the use of type annotations in a program that sums the elements of a 1000 by 1000 matrix:

```
(defvar my-matrix (make-array '(1000 1000)
                    :element-type 'single-float
                    :initial-element 1.0s0))

(defun sum-elts (xs)
  (declare (type (simple-array single-float (1000 1000))
 ↪  xs))
  (let ((sum 0.0s0))
    (declare (type single-float sum))
    (dotimes (i 1000)
      (dotimes (j 1000)
         (incf sum (aref xs i j ))))
    sum))

(format t "~A~%" (sum-elts my-matrix))
```

The type specification in `make-array` allows for a more efficient, unboxed, memory representation for the array and the type declarations inside `sum-elts` allow the compiler to use efficient floating-point arithmetic inside the main loop. According to tests performed by Graham [54, page 221], the version of the program with the type declarations runs an order of magnitude faster than the version without them.

5.2 The effect of type annotations in Common LISP

Common LISP type annotations allow the programmer to specify to the compiler their intent about the types in their code. This mainly affects the program behavior in two ways: error detection and type-directed optimization. In this section we illustrate some of these properties of the Common LISP type annotation system by providing some example programs and observing how they behave. Since the Common LISP language specification does not specify many aspects of the type system, we will not attempt to provide a general semantics for type annotations and will limit ourselves to reporting the behavior of a single Common LISP implementation. We chose Steel Bank Common LISP (SBCL) version 1.3 [55] because it allowed us to showcase undefined behavior and other observable effects of type annotations. All our tests were run on 64 bit Debian Linux.

5.2.1 Error detection

Common LISP implementations are allowed to produce type errors if the type annotations the programmer provides do no match. We can see this in the two programs in Fig. 24, which SBCL 1.3 refuses to run due to static type errors.

Common LISP implementations can also insert runtime type checks to verify the validity of type declarations that are not verifiable at compile time.

Function where Int is expected

```
(defun foo (a b)
  (declare (function a))
  (declare (fixnum b))
  (+ a b)) ;; <- "a" is not a number
```

Int where Function is expected

```
(defun foo (f x)
  (declare (fixnum f))
  (declare (fixnum x))
  (funcall f x)) ;; <- "f" is not a function
```

Figure 24 – Compile-Time Type Errors in Common LISP

```
(defun add1 (x)
  (declare (fixnum x))
  (+ x 1))

(format t "~A~%" (add1 1  )) ;; Prints "2"
(format t "~A~%" (add1 1.0)) ;; Runtime TYPE-ERROR
```

Figure 25 – Runtime Type Errors in Common LISP

This is demonstrated in the program in Fig. 25, where the `(add1 1.0)` function call raises a runtime type error. Note that this error was not detected at compile time because the type declaration for `x` only affects the body of `add1`.

There are some limits to Common LISP's type checking, however. One notable one is that the input and return types are ignored when comparing two function types. As shown in Fig. 26, the type checker considers the types of integer and floating-point numbers to be incompatible but does not do the same when the type declarations are for functions over these incompatible types.

5.2.2 Type-directed optimization

Although Common LISP type declarations can be used for error checking and documentation, these were not the most important reasons why type declarations were added to Common LISP. According to Guy Steele's Overview of Common LISP [52], the primary motivation behind type declarations was their use in type-directed compiler optimizations:

> A type declaration facility is provided by which the user can advise an optimizing compiler. [...] the user undertakes to guarantee that the arguments will be of the specified type. Compiled code may assume this or may perform runtime-checks to confirm the

Contradictory number-type annotations

```
;; These declarations cause a compile-time error
(defun foo (x)
  (declare (fixnum x))
  (declare (single-float x))
  x)
```

Contradictory function-type annotations

```
;; These declarations do not cause any errors
(defun bar (f)
  (declare (ftype (function (fixnum) fixnum) f))
  (declare (ftype (function (single-float) single-float)
 ↪  f))
  f)
```

Figure 26 – Compile-Time Type Errors in Common LISP

> declaration: in any case the compiler may be able to generate more efficient code for the body of the function.

This focus on performance and the willingness to allow undefined behavior in the cases if type annotations are not respected at runtime are the most distinctive characteristics of Common LISP's type annotation system. Since Common LISP implementations are allowed to assume that type declarations are always correct, programs mixing statically-typed and dynamically-typed parts are susceptible to undefined behavior.

One example of such undefined behavior can be seen in Fig. 27. The `add-DD` and `add-FF` functions simply add their two inputs together and the only difference between them is that `add-FF` declares its inputs as floating-point numbers while `add-DD` is dynamically-typed (the default). According to my tests, SBCL 1.3 prints the nonsensical `0.0` when integers are passed to the `add-FF` function. The precise value of this wrong number is not important. What is really important is that its existence indicates that when SBCL is configured with a safety setting of zero the optimizer fully trusts the type annotations and `add-FF` is compiled to blindly add its operands with floating-point arithmetic and to always return a floating-point number.

Undefined behavior is not limited to incorrect arithmetic coercions. The program in Fig. 28 attempts to access an invalid memory address when it treats an integer as if it were a function pointer. The address of the memory fault,`0x7`, is correlated to the value of x. In fact, the address is always equal to $2x - 3$, which likely has to do with the internal tagged representation for fixnums and function pointers in SBCL.

```
(declaim (optimize (safety 0)))

(defun add-DD (x y)
  (+ x y))

(defun add-FF (x y)
  (declare (single-float x y))
  (+ x y))

;; add-DD performs dynamic arithmetic.
(format t "~A~%" (add-DD 1   2   )) ;; 3
(format t "~A~%" (add-DD 1.0 2.0)) ;; 3.0

;; add-FF returns nonsense floating-point numbers if its
 ↪  inputs are not of the declared type.
(format t "~A~%" (add-FF 1   2   )) ;; 0.0 (wrong value)
(format t "~A~%" (add-FF 1.0 2.0)) ;; 3.0
```

Figure 27 – Type Directed Optimization in Common LISP

Program

```
(declaim (optimize (safety 0) (speed 3)))

(defun foo (x)
  (declare (function x))
  (funcall x 0))

(format t "~A~%" (foo 5))
```

Error message

```
CORRUPTION WARNING in SBCL pid 24990(tid 140737353897728):
Memory fault at 0x7 (pc=0x10039e0c6f, sp=0x7ffff2d1ecd0)
The integrity of this image is possibly compromised.
Exiting.
```

Figure 28 – Memory corruption in Common LISP

6 Gradual Typing

It is very hard to design a type system or static analysis tool that can reason about all the idioms that are commonly used in dynamically-typed programming languages [29; 56]. Because of this, one approach that has become very popular recently is Gradual Typing [57–63], which is based on explicitly segregating programs in dynamically-typed and statically-typed sections. This allows the programmer to leave the some parts of the program intentionally untyped and gradually convert untyped code to typed code, one part at a time.

In this chapter, we focus on one of the simplest gradually-typed systems, the gradually-typed λ-calculus of Siek and Taha [57]. In Section 6.1 we cover the blame calculus, a programming language that is ideal for studying the mixing of static and dynamic typing, but which is too verbose to use directly. In Section 6.2 we describe the gradually-typed λ-calculus in terms of the blame calculus and in Section 6.3 we show the most important properties that a gradual type system should follow. Finally, in Section 6.4 we briefly discuss the challenges that need to be faced in order to create gradually-typed languages with type systems that are more powerful than the simply-typed one.

6.1 The Blame Calculus

The central idea of gradual typing is to allow programmers to write part of their code in a dynamically-typed language and another part of their code in a statically-typed dialect of that language, while still having them interact in a sound manner. The meaning of this soundness is something that we will explain soon, in Section 6.1.4.

One of the simplest model languages for this interaction between typed and untyped code is the *blame calculus* of Wadler and Findler [64]. The blame calculus is an intermediate language that makes the static and dynamic parts of the program fully explicit. We will describe it in detail in the following subsections.

6.1.1 Syntax of the Blame Calculus

We show the syntax for the Blame Calculus in Fig. 29. The statically-typed part of the blame calculus is essentially the simply-typed λ-calculus. In our presentation we use **null** and **int** as the base types, as they will suffice for most of the examples we will use. To streamline the semantics and type system we model arithmetic operations as function constants instead of as primitive language operations.

Dynamic typing in the blame calculus is represented explicitly on top of the statically-typed λ-calculus using a variant record, a technique we mentioned in Section 3.1.2. Tagged values have the dynamic type $\star$ and are written $(\mathrm{Dyn}_G v)$. The tag G is a ground type, which can be a base type (**null**

Type syntax

Base Type	B	::=	$\mathbf{null} \mid \mathbf{int}$
Type	T	::=	$B \mid T_1 \to T_2 \mid \star$
Ground Type	G	::=	$B \mid \star \to \star$

Term syntax

Var	x	::=	$x, y, z, \ldots$
Function	f	::=	$\lambda T : x\,.\,e$
Integer	n	::=	$0, 1, 2, \ldots$
Primitive Functions	k	::=	$+, -, \times, \div, \ldots$
Constant	c	::=	$\mathrm{null} \mid 0, 1, 2, \ldots \mid k$
Blame Label	ℓ	::=	$\ell_1, \ell_2, \ldots$
Expr	e	::=	$c \mid x \mid f \mid (\mathrm{Dyn}_G e) \mid (e_1\ e_2) \mid \langle T_2 \Leftarrow T_1 \rangle^{\ell}\ e$

Figure 29 – Syntax of the blame calculus

or **int**) or the type of dynamic functions, $\star \to \star$. These are the types that can be checked dynamically at runtime. The interface between statically-typed and dynamically-typed parts of the language is bridged by explicit casts, written $\langle T_2 \Leftarrow T_1 \rangle^{\ell}$. The label ℓ is unique for each cast and corresponds to their position in the original program source code (line and column number).

Some examples and a lighter notation

The following program adds together a typed integer with another integer that is wrapped inside a dynamic value. The addition operation uses prefix notation because in the blame calculus addition is just a regular function of two parameters.

$$\begin{gathered}\mathbf{let}\, x : \star = (\mathrm{Dyn}_{\mathbf{int}} 1)\ \mathbf{in} \\ (+\ 2\ (\langle \mathbf{int} \Leftarrow \star \rangle^{\ell}\ x))\end{gathered}$$

Note how all the variables are accompanied by explicit type annotations and how there is an explicit difference between untagged, statically-typed numbers (such as 2) and tagged, dynamically-typed numbers (such as $(\mathrm{Dyn}_{\mathbf{int}} 1)$). Function calls are statically-typed and dynamic values must be cast down to untagged integers before they can be passed to arithmetic operations.

Since writing down all the Dyns and casts from $\star$ is very notationally heavy, we will use Wadler's $\lceil\ \rceil$ notation to embed a more familiar-looking untyped λ-calculus inside the blame calculus. The precise meaning of $\lceil\ \rceil$ notation is described in Fig. 30. The $\lfloor\ \rfloor$ notation is used as a form of quasi-quoting to embed typed blame calculus expressions inside the untyped expressions. Bracket notation is specially space-saving when there are many function calls in the dynamically-typed parts of the code, as can be seen in the examples in Fig. 31. Notice how a cast needs to be inserted to convert the typed *inc* function into a dynamic value and how that dynamic value needs to be casted back to $\star \to \star$ when its called, to check if its really a function.

$$\begin{aligned}
\lceil x \rceil &= x \\
\lceil \text{null} \rceil &= (\text{Dyn}_{\mathbf{null}} \text{null}) \\
\lceil n \rceil &= (\text{Dyn}_{\mathbf{int}} n) \\
\lceil + \rceil &= (\text{Dyn}_{\star \to \star} \langle \star \to \star \to \star \Leftarrow \mathbf{int} \to \mathbf{int} \to \mathbf{int} \rangle\ +) \\
\lceil \lambda x.\, e \rceil &= (\text{Dyn}_{\star \to \star} \lambda x : \star\, .\, \lceil e \rceil) \\
\lceil (e_1\ e_2) \rceil &= ((\langle \star \to \star \Leftarrow \star \rangle\ \lceil e_1 \rceil)\ \lceil e_2 \rceil) \\
\lceil \lfloor e \rfloor \rceil &= (\text{Dyn}_G \langle G \Leftarrow T \rangle\ e) \qquad \text{if } e : T \text{ and } G \sim T
\end{aligned}$$

Figure 30 – Wadler's $\lceil\rceil$ notation

A program using $\lceil\rceil$ notation

$$\begin{aligned}
&\mathbf{let}\ inc = \lambda x : \mathbf{int}.\ (+\ x\ 1)\ \mathbf{in} \\
&\lceil (\lfloor inc \rfloor\ \text{null}) \rceil
\end{aligned}$$

A program without $\lceil\rceil$ notation

$$\begin{aligned}
&\mathbf{let}\ inc = \lambda x.\ (+\ x\ 1)\ \mathbf{in} \\
&((\langle \star \to \star \Leftarrow \star \rangle\ (\text{Dyn}_{\star \to \star} \langle \star \to \star \Leftarrow \mathbf{int} \to \mathbf{int} \rangle\ int))\ (\text{Dyn}_{\mathbf{null}} \text{null}))
\end{aligned}$$

Figure 31 – Example of $\lceil\rceil$ notation

6.1.2 The Static Type System of the Blame Calculus

In Fig. 32 we show the typing rules for the blame calculus. The first group of rules – VAR, CONST, LAM and APP – is responsible for typing the static parts of blame calculus programs and is lifted straight from the simply-typed λ-calculus. The only changes compared to the simply-typed calculus we presented in Section 2.4 is that the CONST rule unifies the various rules for typing literals under a single rule. The meta function TypeOf returns the type of a given constant. For example, $\text{Typeof}(1) = \mathbf{int}$ and $\text{Typeof}(+) = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$.

The second kind of rule is the DYN rule, which says that dynamic values have the type $\star$. Note that the $\star$ type is just a regular variant type, except that we use casts instead of **case** expressions to extract fields from it. In particular, the $\star$ type is *not* a supertype of the other types, as would be found in a type system with subtyping. This is a common source of confusion for people being introduced to gradual typing, specially since many gradually-typed languages call the $\star$ type by **any**, which suggests a set-subtyping relation.

Finally, the CAST rule governs the transition between the typed and untyped parts of the programs. The CAST rule refers to a *type consistency relation*, written $\sim$. $S \sim T$ means that it is possible to make the types S and T equal to one another by replacing all the occurrences of the dynamic type $\star$ in S and T

Typing rules $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\,\text{VAR} \qquad \frac{}{\Gamma \vdash c : \text{TypeOf}(c)}\,\text{CONST} \qquad \frac{\Gamma[x \leftarrow S] \vdash e : T}{\Gamma \vdash \lambda x : S\,.\,e : S \to T}\,\text{LAM}$$

$$\frac{\Gamma \vdash e_1 : S \to T \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash (e_1\; e_2) : T}\,\text{APP} \qquad \frac{\Gamma \vdash e : G}{\Gamma \vdash (\text{Dyn}_G e) : \star}\,\text{DYN} \qquad \frac{\Gamma \vdash e : S \qquad S \sim T}{\Gamma \vdash \langle T \Leftarrow S \rangle^{\ell}\, e : T}\,\text{CAST}$$

Type consistency $\boxed{S \sim T}$

$$\mathbf{null} \sim \mathbf{null} \qquad \mathbf{int} \sim \mathbf{int} \qquad T \sim \star \qquad \star \sim T \qquad \frac{S \sim S' \qquad T \sim T'}{S \to T \quad S' \to T'}$$

Figure 32 – Type system for the blame calculus

with appropriate types that do not contain $\star$.

The consistency relation rules out *foolish* casts at compile time, such as $\langle \mathbf{int} \Leftarrow \mathbf{null} \rangle$. The only way to write a program that tries to use a **null** as if it were an **int** is by casting to the dynamic type as an intermediate step: $\langle \mathbf{int} \Leftarrow \star \rangle\ \langle \star \Leftarrow \text{null} \rangle$.

Type consistency is reflexive and symmetric but is not transitive. For example, we have $\mathbf{null} \sim \star$ and $\star \sim \mathbf{int}$ but $\mathbf{null} \nsim \mathbf{int}$. Another thing to note is that the consistency relation is not a subtyping relation. $\sim$ is covariant in the first parameter of function types, unlike subtyping relations, which are contravariant at that position.

6.1.3 Reduction Semantics of the Blame Calculus

The reduction semantics of the blame calculus is shown in Fig. 33. The evaluation of the statically-typed parts is not surprising and the rules are the same as they are in the simply-typed λ-calculus. The $[\![k]\!]$ in the primop rule is the mathematical function that denotes the meaning of the function constant. For example, $[\![+]\!](1, 2) = 3$.

The interesting part of the blame calculus semantics is in the rules governing the casts. In addition to the trivial id-cast rule, there are three kinds of rule for casts: the rules for casting to $\star$, the rules for casting from $\star$, and the rule for casting between function types.

Casting to the dynamic type

According to the base-to-dyn rule, casting a value of a base type to $\star$ consists of wrapping it inside a tagged Dyn. The rule for casting functions to $\star$, fun-to-dyn is slightly more complicated. We must first cast the function to $\star \to \star$ if it does not already have that type, because this is the only function type that can be tested for by simply checking the type tag of the value. The importance of this added cast will become clearer when we discuss casts between function types.

Casting from the dynamic type

The rules governing casts from the dynamic type are the DYN-TO-T and CAST-ERROR rules. As suggested by the name of the second rule, these casts from the dynamic type are where runtime errors can possibly come from. The result of a cast $\langle T \Leftarrow \star\rangle^{\ell}$ $(\mathrm{Dyn}_G v)$ will depend on whether the runtime type tag G is compatible with the desired result type T.

If $G \nsim T$, we abort the evaluation of the program, returning the error **blame** ℓ. This error message contains the line and column number of the cast in the original program that is responsible for the error that just occurred. This *blame tracking* is important because stack traces are not sufficient to pinpoint the true source of runtime type errors in the blame calculus. As we will see later, casts between function types report their type errors when the casted function is applied, not when the function cast is evaluated.

On the other hand, if $G \sim T$, the cast does not immediately fail. If T is a base type or the function type $\star \to \star$, then the cast succeeds and returns the unwrapped value, after applying the ID-CAST rule once. On the other hand, if T is a function type that is not $\star \to \star$, then it is not possible to immediately detect if the casts succeed or not. Instead, it is converted to the cast $\langle T \Leftarrow \star \to *\rangle^{\ell}\, v$, which may fail at a later date, as will be explained when we describe function casts. Note that the new cast contains the location information ℓ from the old cast, so the error message will point to the correct line and column number in case of a runtime type error.

Casts between function types

The only casts that are neither to or from the $\star$ type and that still respect the consistency rule discussed in Section 6.1.2 are casts where a base type is cast to itself or casts between function types. The first kind of cast succeeds trivially. The second kind of cast is not so simple because it is impossible to immediately tell if doing the cast will result in a runtime error. For example, consider the following program, where a typed function g is given an untyped implementation f:

$$\begin{aligned}&\mathbf{let}\, f : \star \to \star = \lambda y.\, \lceil(+\ 1\ \lfloor y \rfloor)\rceil\ \mathbf{in}\\ &\mathbf{let}\, g = \langle \mathbf{int} \to \mathbf{int} \Leftarrow \star \to \star\rangle^{\ell} f\ \mathbf{in}\\ &(g\ 2)\end{aligned}$$

By inserting the $\langle \mathbf{int} \to \mathbf{int} \Leftarrow \star \to \star\rangle\, f$ cast, the programmer is asserting that if the untyped function f receives an input that is an integer, then it will return an output that is an integer. However, this assertion is undecidable at compile time so the blame calculus must check at runtime that this belief from the programmer actually holds. As shown in rule FUN-TO-FUN, function casts are evaluated into wrapper functions that perform runtime checks on the inputs and outputs. In our case it will look like this:

$$\begin{aligned}&\langle \mathbf{int} \to \mathbf{int} \Leftarrow \star \to \star\rangle^{\ell} f \longmapsto\\ &\lambda x : \mathbf{int}.\, \langle \mathbf{int} \Leftarrow \star\rangle^{\ell}\ (f\ (\langle \star \Leftarrow \mathbf{int}\rangle^{\ell}\ x))\end{aligned}$$

Note the presence of the $\langle \mathbf{int} \Leftarrow \star\rangle^{\ell}$ cast just before the wrapper returns. If f ever returns a value that is not an integer then the program will be aborted

Irreducible expressions

$$\begin{array}{llll} \text{Value} & v & ::= & c \mid f \mid (\mathrm{Dyn}_G v) \\ \text{Result} & r & ::= & v \mid \textbf{blame}\ \ell \end{array}$$

Reduction contexts

$$\text{Reduction context} \quad C \quad ::= \quad [\,] \mid (\mathrm{Dyn}_G C) \mid (C\ e) \mid (v\ C) \mid \langle T_2 \Leftarrow T_1 \rangle^\ell\, C$$

Reduction rules $\boxed{e \to e \cup \textbf{blame}\ \ell}$

$$\begin{array}{rll} C[((\lambda x : T\,.\,e)\ v)] & \longmapsto C[e[x \leftarrow v]] & (\textsc{app}) \\ C[(k\ v)] & \longmapsto C[[\![k]\!](v)] & (\textsc{primop}) \\ C[\langle T \Leftarrow T \rangle^\ell\, v] & \longmapsto C[v] & (\textsc{id-cast}) \\ \langle S' \to T' \Leftarrow S \to T \rangle^\ell\, v & \longmapsto \lambda x : S'\,.\,\langle T' \Leftarrow T \rangle^\ell\, (v\ (\langle S \Leftarrow S' \rangle^\ell\, x)) & (\textsc{fun-to-fun}) \\ & \quad \text{with } x \notin FV(v) & \\ C[\langle \star \Leftarrow B \rangle^\ell\, v] & \longmapsto C[(\mathrm{Dyn}_B v)] & (\textsc{base-to-dyn}) \\ C[\langle \star \Leftarrow S \to T \rangle^\ell\, v] & \longmapsto C[(\mathrm{Dyn}_{\star\to\star} \langle \star \to \star \Leftarrow S \to T \rangle^\ell\, v)] & (\textsc{fun-to-dyn}) \\ C[\langle T \Leftarrow \star \rangle^\ell\, (\mathrm{Dyn}_G v)] & \longmapsto C[\langle T \Leftarrow G \rangle^\ell\, v] \qquad \text{if } G \sim T & (\textsc{dyn-to-t}) \\ C[\langle T \Leftarrow \star \rangle^\ell\, (\mathrm{Dyn}_G v)] & \longmapsto \textbf{blame}\ \ell \qquad \text{if } G \nsim T & (\textsc{cast-error}) \end{array}$$

Figure 33 – Reduction semantics for the blame calculus

and the original cast will be the one blamed because the casts in the wrapper function contain the location information of the original cast. This *blame tracking* means that the error messages for type errors do not point to the place in the program where they occur, as typically happens when a dynamic language aborts and shows a stack trace. Instead, type errors point toward the cast whose assertion was violated.

The cast at the start of the wrapper function is for when we pass a typed function to an untyped context. For example, in the following program we have a $\langle \star \Leftarrow \textbf{int} \to \textbf{int} \rangle$ cast instead of a $\langle \textbf{int} \to \textbf{int} \Leftarrow \star \to \star \rangle$ one.

$$\begin{array}{l} \textbf{let}\, f : \textbf{int} \to \textbf{int} = \lambda y.\,(+\ 1\ y)\ \textbf{in} \\ \textbf{let}\, g : \star = \langle \star \Leftarrow \textbf{int} \to \textbf{int} \rangle^\ell f\ \textbf{in} \\ \lceil (g\ 2) \rceil \end{array}$$

This time, the assertion the programmer is making is that the untyped function g will always receive inputs that are integers. The purpose of the check at the start of the wrapper created by the function-type cast is to detect at runtime if this assumption is violated.

6.1.4 Soundness of the blame calculus

The most basic statement of soundness for the blame calculus is the same kind of statement that is made for the simply-typed λ-calculus and for the dynamically-typed λ-calculus: well-typed programs don't go wrong. In the blame calculus this translates to the following type safety theorem:

Theorem (Type-safety of the Blame Calculus)**.** *Let e be a well-typed term in the blame calculus. One of the following holds:*

- $e \to^* v$
- $e \to^*$ ***blame*** *ℓ, for some blame label ℓ in e*
- *the evaluation of e diverges*

A proof of this type safety can be found in most papers introducing a gradual type system. For instance, Siek and Taha's original gradual typing paper [57].

There are also other more advanced soundness theorems that we do not ennunciate formally also assign a direction to the blame in addition to a location [65; 66]. Informally, when a cast error occurs in a function call, blame may be assigned either to the caller (which passed an invalid value to the function) or to the function (which returned an invalid result to the caller). With this sense of direction these soundness theorems manage to say that in a program mixing statically-typed and untyped code, blame will always lie on the untyped side.

6.2 A Surface Syntax for the Blame Calculus

Wadler's blame calculus works very well as a low-level language for mixing statically-typed and dynamically-typed programming. However it is too verbose to use on a day to day basis. In this section, we present Siek and Taha's gradually-typed λ-calculus [57], which is a programmer-facing syntax for the blame calculus.

Fig. 34 describes the syntax of the gradually-typed λ-calculus and a set of rules for converting gradually-typed programs into the blame calculus. These rules simultaneously fulfill the role of a type system and a semantics. A gradually-typed program is considered well-typed if it can be converted to the blame calculus. Evaluating a gradually-typed program consists of converting it to its blame calculus equivalent and evaluating that program.

Marking sections as typed or untyped in the gradually-typed calculus is just a matter of adding optional type annotations to variable declarations. Omitting a type declaration by writing $(\lambda x.\, e)$ is equivalent to writing $(\lambda x : \star\,.\, e)$. If a variable declaration is annotated then that variable is statically-typed and if a variable has no type annotation then it has the dynamic type $\star$. Other gradually-typed languages may use type inference to decide the type of unannotated variables instead of always using $\star$. However, this can result in the type system eagerly rejecting a program that would have executed correctly had the variables been typed with $\star$.

The more interesting rules in the conversion process are the rules for converting function applications. These are the rules that insert the type casts that the blame calculus expects, which is also why the function applications

Syntax of the surface language

Var	x	::=	$x, y, z, \ldots$
Function	f	::=	$\lambda x : T \,.\, e$
Integer	n	::=	$0, 1, 2, \ldots$
Primitive Functions	k	::=	$+, -, \times, \div, \ldots$
Constant	c	::=	$\mathrm{null} \mid 0, 1, 2, \ldots \mid k$
Blame Label	ℓ	::=	$\ell_1, \ell_2, \ldots$
Expr	e	::=	$c \mid x \mid f \mid (e_1\ e_2)^{\ell}$

Optional type annotations

$\lambda x.\, e$ is syntactic sugar for $\lambda x : \star \,.\, e$.

Conversion to the blame calculus $\boxed{\Gamma \vdash e \hookrightarrow e' : T}$

$$\textsc{var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x \hookrightarrow x : T} \qquad \textsc{const}\ \frac{\mathrm{TypeOf}(c) = T}{\Gamma \vdash c \hookrightarrow c : T} \qquad \textsc{lam}\ \frac{\Gamma[x \leftarrow S] \vdash e \hookrightarrow e' : T}{\Gamma \vdash \lambda x : S \,.\, e \hookrightarrow \lambda x : S \,.\, e' : (S \rightarrow T)}$$

$$\textsc{app-no-check}\ \frac{\Gamma \vdash e_1 \hookrightarrow e_1' : (S \rightarrow T) \qquad \Gamma \vdash e_2 \hookrightarrow e_2' : S}{\Gamma \vdash (e_1\ e_2)^{\ell} \hookrightarrow (e_1\ e_2) : T}$$

$$\textsc{app-check-arg}\ \frac{\Gamma \vdash e_1 \hookrightarrow e_1' : (S \rightarrow T) \qquad \Gamma \vdash e_2 \hookrightarrow e_2' : S' \qquad S \neq S' \qquad S \sim S'}{\Gamma \vdash (e_1\ e_2)^{\ell} \hookrightarrow (e_1\ (\langle S \Leftarrow S' \rangle^{\ell}\, e_2)) : T}$$

$$\textsc{app-check-fun}\ \frac{\Gamma \vdash e_1 \hookrightarrow e_1' : \star \qquad \Gamma \vdash e_2 \hookrightarrow e_2' : S}{\Gamma \vdash (e_1\ e_2)^{\ell} \hookrightarrow ((\langle S \rightarrow \star \Leftarrow \star \rangle^{\ell}\, e_1)\ e_2) : \star}$$

Figure 34 – A high-level syntax convertible to the blame calculus

are identified by labels, for blame-tracking purposes. There are two cases that matter when converting a function application $(f\ x)^{\ell}$.

The first case is when the type of f is a function type. In this case, a cast is inserted to convert the input to the domain type of the function. If the two types are equal then the application is fully statically-typed and the cast is not actually needed, as shown in Rule APP-NO-CHECK. Otherwise, Rule APP-CHECK-ARG applies.

The second possibility for function applications is that the function has the dynamic type $\star$. In this case, the cast is inserted at the function instead of at the parameter, per the APP-CHECK-FUNC rule.

If the type of the function parameter is a base type then the program cannot be converted to the blame calculus and is considered ill-typed. Similarly, a program where a function receives an input that is not compatible with its domain type according to the $\sim$ relation is also ill-typed.

6.3 Properties of Gradually-Typed Systems

In Section 6.2 we describe a gradually-typed programming language but we do not say much about what can be deduced from its type system and evaluation semantics. In this section, we will fill this void by describing three properties that define what it means to be gradually-typed. These properties are the simulation of fully-typed and fully-untyped calculi, the soundness of the blame tracking, and the *gradual guarantee*, that describes what happens as type annotations are added or removed from a program [67].

6.3.1 Gradual Typing is a Superset of Static and Dynamic Typing

The first important property of a gradually-typed language is that for fully annotated programs it should behave like a statically-typed language, and for fully un-annotated programs it would behave like a dynamically-typed language. If a program is fully annotated with type declarations, evaluating it should never result in a runtime type error. On the flip side, if the program has no type annotations, then it should be as flexible as a dynamic calculus.

In the case of the gradually-typed λ-calculus of Siek and Taha, this means that fully annotated programs should behave exactly as the simply-typed λ-calculus and fully un-annotated programs should behave exactly like the dynamically-typed λ-calculus.

The fully annotated case can be formalized in the following theorem. $\vdash_G$ and $\vdash_S$ are the typing judgments for the gradually-typed and simply-typed calculi, respectively. $\Downarrow_G$ and $\Downarrow_S$ are their big-step evaluation relations.

Theorem (Equivalence to the simply-typed λ-calculus)**.** *For every program e in the statically-typed λ-calculus it holds that:*

- $\vdash_G e : T$ *if and only if* $\vdash_S e : T$
- $e \Downarrow_G v$ *if and only if* $e \Downarrow_S v$

The proof of this theorem is easy to demonstrate. When every term is annotated with static types that do not contain $\star$, the translation from the gradually-typed λ-calculus to the blame calculus never inserts any type casts (as is determined by the APP-NO-CHECK rule). The fragment of the blame calculus without any type casts has precisely the same evaluation and typing rules as the simply-typed λ-calculus.

The equivalence of the dynamically-typed case is slightly more subtle. We must convert all literals to the dynamic type, to avoid ever raising a static type error. Since our syntax for the surface language does have a cast operator to convert numeric literals to the $\star$ type, the theorem instead shows the equivalence between the dynamically-typed λ-calculus and the dynamic fragment of the blame calculus. The $\lceil\ \rceil$ operation is the same one described in Section 6.1. $\Downarrow_B$ and $\Downarrow_D$ are the big step evaluation relations for the blame calculus and the dynamically-typed λ-calculus, respectively.

Theorem (Equivalence to the dynamically-typed λ-calculus)**.** *For every program e in the dynamically-typed λ-calculus it holds that:*

- $\vdash_B \lceil e \rceil : \star$
- *$e \Downarrow_B v$ if and only if $e \Downarrow_D v$*

The proof of this theorem is slightly more involved than the proof for the static case and we omit it for brevity. It can be found in Siek and Taha's original gradual typing paper [57].

6.3.2 Soundness of the Interaction Between Static and Dynamic Typing

As we previously discussed in Section 6.1.4, a gradually-typed system should be type sound and it should also guarantee that the more typed parts of the program are not to blame for any type errors.

The type safety ensures that the interaction between typed and untyped code never results in undefined behavior. The purpose of the blame tracking is to guarantee that the types written by the programmer are respected. This is what separates full gradually-typed systems from optionally typed systems that use a similar type system with dynamic types and the consistency relation, but without adding runtime checks to the boundaries between typed and untyped code.

6.3.3 The Gradual Guarantee

The third final important property of a gradually-typed system concerns with how program behaves after after type annotations are added to them. The main objective of gradual typing is to allow untyped programs to be gradually transitioned into typed programs and it is important that this process can happen in a sound and predictable manner.

Adding a type annotation should never change the result evaluating the program, except that the extra type annotation may cause the program to start raising a runtime error it did not raise before. If the program with the additional type annotation does not raise a runtime type error when evaluated then its result should be the same as the program without the type annotation.

To be able to formally formulate the Gradual Guarantee, we first define what it means for a program to be more precisely typed than another. In Fig. 35, we define a pair of relations $T_1 \sqsubseteq T_2$ and $e_1 \sqsubseteq e_2$ that define what it means for a type T_1 to be more static than T_2 and what it means for a program e_1 to be more statically-typed than another program e_2.

With the $\sqsubseteq$ relations at our disposal, we define the Gradual Guarantee as follows:

Theorem (Gradual Guarantee). *Given a pair of programs e and e', where $e \sqsubseteq e'$ and $\vdash e : T$, the following hold:*

- *$\vdash e' : T'$ and $T \sqsubseteq T'$*
- *If $e \Downarrow v$ then $e' \Downarrow v'$ and $v \sqsubseteq v'$*
- *If e diverges then e' diverges.*

Type Precision $\boxed{T_1 \sqsubseteq T_2}$

$$\textbf{null} \sqsubseteq \textbf{null} \qquad \textbf{int} \sqsubseteq \textbf{int} \qquad T \sqsubseteq \star \qquad \frac{S \sqsubseteq S' \qquad T \sqsubseteq T'}{S \to T \sqsubseteq S' \to T'}$$

Term Precision $\boxed{e_1 \sqsubseteq e_2}$

$$x \sqsubseteq x \qquad c \sqsubseteq c \qquad \frac{T_1 \sqsubseteq T_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : T_1 \,.\, e_1 \sqsubseteq \lambda x : T_2 \,.\, e_2} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1\ e_2)^\ell \sqsubseteq (e_1'\ e_2')^\ell}$$

Figure 35 – Type-precision relations

- *if $e' \Downarrow v'$ then either $e \Downarrow v$, with $v \sqsubseteq v'$ or $e \Downarrow$ **blame** ℓ*
- *if e' diverges then either e diverges or $e \Downarrow$ **blame** ℓ*

It is possible to show, using the Blame Theorem, that the gradually-typed λ-calculus offers the Gradual Guarantee. However, for languages with more complex type systems, it is not hard to violate this guarantee, even if the language designer has the best intentions from the start.

One example of a language feature that would violate the Gradual Guarantee is a try-catch statement that can handle runtime type errors. If the runtime errors from the type casts don't immediately abort the execution of the program it is possible to write a typed program that evaluates to something different than its untyped equivalent.

A second example of a language feature that can violate the Gradual Guarantee is a === operator for equality testing via object-identity. With this operation, it might be possible to tell apart a function and a version of the same function that is wrapped by a higher-order cast [68].

6.4 Challenges for Gradual-Typing

The gradually-typed calculus of Siek and Taha guarantees that blame is assigned in a sound manner and also upholds the Gradual Guarantee. However, doing this is not as easy once the gradually-typed language starts having a type system that is more advanced than the simply-typed one. And in addition to the problem of just providing blame tracking for advanced type system features, there is also the problem of doing so in a performant manner.

One example of a type system feature that is not easy to integrate with gradual typing is parametric polymorphism. In a statically-typed language, the type system can guarantee relational parametricity (theorems for free) at compilation time and erase all type information during execution. On the other hand, a gradually-typed system must provide some way to dynamically enforce relational parametricity, in case a statically-typed function with parametric types is given a dynamically-typed implementation. Ahmed et al show that it is possible to do this by inserting opaque wrappers around values with parametric types [69]. However, it is not yet clear if these wrappers are viable in practical languages.

Mutable objects and data structures are another example of a language feature that is not easy to implement in a gradually-typed setting. To prevent object invariants from being violated by dynamically-typed code, statically-typed objects that are passed to dynamically-typed functions must be wrapped in a wrapper that only allows the dynamically-typed code to perform reads, not writes.

Recursive types also present a problem. It only takes a single tag check to verify whether a dynamic value is an integer but it might take an arbitrarily large number of checks to verify that a dynamic value is a list of integers.

In an effort to sidestep this issue, some gradually-typed languages sacrifice the expression-level granularity present in Siek's gradually-typed calculus in exchange for simplifying the blame tracking. The poster child of this approach is the Typed Racket programming language [65; 70]. In Typed Racket, gradual typing happens at the granularity of whole modules. Typed Racket modules can interact with regular untyped Racket modules but a single module must be either fully typed or fully untyped. What Typed Racket gains from this restriction is that the typed modules are allowed to use many type system features that are hard to gradually-type, such as parametric polymorphism.

7 Optional Typing

In Chapter 5 and Chapter 6 we had a look at type systems based on optional type annotations, both of which use type annotations to influence the runtime behavior of the program. In Common LISP's case the type annotations aid aggressive compiler optimizations (which are observable when undefined behavior is triggered) while in gradually-typed languages the type annotations are used to insert powerful runtime assertions and blame tracking. However, there is a third path that can be taken in a system with type annotations, which is to simply ignore them at runtime and only use them for tooling and other compiler diagnostics. This approach is commonly known as *Optional Typing*, a term coined by Gilad Bracha [71], whose Strongtalk [72] was one of the first type systems intentionally designed to have type annotations that do not affect the program at runtime.

In this chapter we discuss Optionally Typed systems for dynamic languages and why their designers chose to not have their type annotations affect the program runtime. We start with a brief overview of Strongtalk in Section 7.1. In Section 7.2 we discuss more recent Optionally Typed languages whose type systems are inspired by Gradual Typing's type-consistency relation but which intentionally do not attempt to perform blame tracking.

7.1 Optional Typing in Strongtalk

The original Optional Typing system is the type system of Strongtalk [72], a commercial Smalltalk dialect from Longview Technologies. Although Strongtalk development was halted before it was publicly released (its development team was hired by Sun Microsystems to work in the Java VM), the papers and presentations about its type systems were some of the earliest to feature this style of optional type signatures.

In Strongtalk, type annotations are completely erased before the program executes and therefore have no effect on the semantics of the program. In principle, fully-annotated Strongtalk programs are guaranteed to never "go wrong" with a "method missing" error, although there is no formal proof of that. The behavior of partially-typed programs is defined to be the same behavior as the unannotated version of the program. This means that as long as there is a single untyped class or module in the whole program, the type system cannot offer any hard guarantees for the statically typed sections of the code.

However, an optional type system can still be useful even when it is not producing strong guarantees about the behavior of the program. Some of the advantages of types that we mentioned in Section 1.1, such as documentation and IDE support, apply even if the type system is not fully sound.

7.2 Optional Typing inspired by Gradual Typing

One style of optionally-typed type system that has become relatively popular recently are type systems inspired by gradually-typed systems, which have a dynamic type $\star$ type and an accompanying consistency relation $\sim$, etc). The difference is that these optional type systems only check types at compile time and do not attempt to perform additional type checking or blame tracking at compile time. This can be done for performance reasons (efficient blame tracking is still an open problem) or also due to the optional type system being unsound in the first place (which would make blame tracking pointless).

Examples of optionally-typed systems in the gradual-typing style are TypeScript [62] (a Javascript dialect), Typed Lua [73] and Dart [74]. In this section we will focus our discussion on Typescript, which is notable for its popularity and relative success. Typescrypt is already being used in production in many Javascript projects, such as the popular Angular front-end framework [75] and some IDEs such as Microsoft's Visual Studio [76] already use its types to power auto-completion and other features.

7.2.1 Error checking and Type erasure in Typescript

Similarly to traditional gradually-typed type systems, the Typescript compiler produces compile-time warnings for static types that do not match. This can be seen in the program in Fig. 36. Dynamically-typed values can also be freely passed to statically-typed functions as we demonstrate in the program in Fig. 37, which compiles successfully with no warnings. However, unlike in traditional gradually-typed systems, the type declarations in this second program are ignored at run-time. The resulting error message and stack trace points to the method call in Line 2 instead of to the function call in Line 6.

Type erasure in Typescript is an intentional design choice. One reason for this is that Typescript programs must be compiled down to regular Javascript (the only programming language with universal web-browser support) and it would be hard to implement efficient checking for higher-order contracts without support from the language runtime. Additionally, the main goal of Typescript is to support and enhance existing Javascript development and according to the developers [62], richer runtime type-checking was not their highest priority:

> The types of a TypeScript program leave no trace in the JavaScript emitted by the compiler. There are no run-time representations of types, and hence no run-time type checking. Current dynamic techniques for "type checking" in JavaScript programs, such as checking for the presence of certain properties, or the values of certain strings, may not be perfect, but good enough.

7.2.2 Unsound Types in Typescript

One interesting aspect of Typescript's type system is that it includes many unsound features, such as downcasting, covariance of mutable properties, and string-based indexing of objects. We illustrate this in Fig. 38 with a classic

Program

```
function hello(s:string){
    alert(s.toUpperCase());
}

hello(10);
```

Compile time error message

```
Line 5: Argument of type 'number' is not assignable to
parameter of type 'string'
```

Figure 36 – A compile-time error in Typescript

Program

```
function hello(s:string){
    alert(s.toUpperCase());
}

var n:any = 10;
hello(n);
```

Runtime error message

```
Line 2: Uncaught TypeError: s.toUpperCase is not a
function
```

Figure 37 – A run-time type error in Typescript

array-covariance example. Typescript allows the covariant assignment in Line 9, which makes it possible to indirectly add a Dog to an array of Cats. This breaks the property access in Line 12, which assumes that objects in the `cats` array have the `meow` property.

The presence of these unsound features is intentional. Since Typescript types are erased, unsoundness is not as dangerous as in a system that does type-directed optimizations (where unsoundness can result in undefined behavior). The worst that can happen in a Typescript program is that it will fail like a typical Javascript program would. Additionally, the design of Typescript prioritizes the tooling and documentation aspect of static type checking, which does not depend on soundness as much. Finally, although unsound type systems cannot guarantee that well-typed programs don't go "wrong", they can still be useful for early error detection.

```
interface Animal { }
interface Dog { bark: string }
interface Cat { meow: string }

var felix: Cat = {meow:"mrooow"}
var pluto: Dog = {bark:"woof woof"}

var cats : [Cat] = [];
var animals : [Animal] = cats;
animals[0] = pluto;

var s:string = cats[0].meow;
alert(s);
```

Figure 38 – Unsound array covariance in Typescript

8 Conclusion

In this section, we compare the various type systems we covered in this text from the point of view of the programmer and the type system designer. The programmer is primarily concerned with what aspects of static typing the type system lets him take advantage of while the type system designer has to focus on the design trade-offs of programing language and type-system design.

To recapitulate, the type systems we will be comparing are the Type Specifiers for Common LISP, Optional and Gradual Typing and two variations of Soft Typing: traditional Soft Typing and Success Typing. Although in theory all soft-typing systems operate similarly (by classifying operations as provably-safe, potentially-unsafe and provably-unsafe) it is worth treating success typing separately because the different error reporting strategy has a big effect on how the type system is designed and used.

8.1 Programmer-oriented comparison of type systems

In this section we compare the type systems for dynamic languages using as criteria the static-typing advantages we listed in Section 1.1. In Fig. 39 we subjectively rate how each type system fares on each category. We explain our reasoning in the following paragraphs.

Static Error Checking

Does this type system help detect compile-time errors? Most static type systems for dynamic language are developed with the primary purpose of detecting errors at compilation time but Common LISP's type specifier system is a notable exception. While some Common LISP implementations, such as

	LISP	Optional	Gradual	Soft	Success
Static Errors	?	+	+	+	+
Documentation		++	++		+
Abstraction		+	++		
Efficiency	++		-	+	

Legend

++	Much better than untyped
+	Better than untyped
	Same as untyped
-	Worse than untyped
?	Depends on the implementation

Figure 39 – Comparison of static-typing benefits

SBCL, use the annotations to guide compile-time and run-time type checking, this is not universal and many Common LISP implementations never issue compile-time warnings for mismatched type annotations.

Documentation

Does the type system help document program interfaces and the contents of variables? Type systems designed around optional type annotations, such as Optional Typing and Gradual Typing are ideal for using types as documentation. The exception is the Common LISP type system: since type annotations can allow unsafe program optimizations, the programmer is discouraged from using them unless they are really needed.

Type systems designed around type inference are less helpful when it comes to documentation. In theory, traditional Soft Typing and Success Typing should score the same in this category, since both work by categorizing expressions as provably-safe, potentially unsafe and provably safe. However, traditional Soft Typing encourages having a more complex and expressive type system, which results in less readable inferred types.

Abstraction

Can the programmer define his own types and type abstractions? Again, type systems designed around type annotations have an edge in this category. We give a higher rating to Gradual Typing because not only is it possible to create new type abstractions but the inserted run-time checks and blame tracking can be used to enforce that these abstractions are respected. As for Common LISP, the type specifiers are usually used to speed up operations involving primitive types (numbers, arrays, etc) and do not encourage programmers to define their own abstractions.

Efficiency

Do types help compiler optimizations? Common LISP stands out here, because its type-directed optimizations are predictable and can be reliably used for performance tuning. For the remaining languages, which are not willing to sacrifice the safety of dynamic typing, the answer depends mostly on the soundness of the type system.

Optional Typing systems do not have efficiency as a primary concern and often feature unsound type systems, which are unsuited for program optimization.

While Gradual Typing does allow for optimization inside statically-typed sections of the program, in practice the run-time checks inserted between dynamic and static code add significant overhead [77]. Reducing this overhead and making gradually-typed programs competitive with untyped programs in terms of performance is still an open research problem.

Soft typing accurately predicts what operations in the program are provably safe and has a long tradition of being used for compiler optimization, even when the type system is not exposed to the programmer [78].

	LISP	Optional	Gradual	Soft	Success
Sound			✓	✓	
Flexible		✓		✓	✓
Simple	✓	?	?		✓

Figure 40 – Type system design trade-offs

Finally, Success Typing often infers types that are "too general", frequently falling back to the **any** type. These types are not suitable for optimization purposes, although the underlying type algorithm might, if it differentiates between provably-safe and potentially-safe operations.

8.2 Designer-oriented comparison of type systems

Anecdotally, type system design usually needs to balance three different desirable properties: correctness, expressiveness and simplicity. A correct, or sound, type system can guarantee that well-typed programs do not "go wrong" (for some definition of "go wrong"). Expressive type systems can type a greater variety of programs and finally, simple type systems are easy to use and learn. Unfortunately, in practice it is very difficult to achieve these three goals simultaneously and type systems designers must sacrifice one or more of them.

In this section we again subjectively rate the type systems we covered in previous chapters but this time we focus on soundness, flexibility and simplicity, which are language and type system design concerns. A summary of our rating can be found in Fig. 40.

Common LISP

The Common LISP type system sacrifices soundness for the simplicity and predictability of being able to know that a type annotation will always result in operations being optimized to type-specific versions. The type system assumes that the programmer knows what he is doing with the type declarations and fully trusts them.

As for flexibility, the Common LISP type system was not designed to statically type whole programs. The type system has a very rich set of primitive types (numbers of various sizes, arrays, etc) but is lacking in flexibility-oriented features, such as parametric polymorphism.

Optional and Gradual Typing

There is a large range of type systems that could falls under the umbrella of Optional or Gradual typing, ranging from very symple type systems to very complex ones. Because of this variability, we think it is more prudent to not grade these type systems in the Simplicity criterion.

When it comes to Soundness vs Flexibility, Optional and Gradual typing take different sides of the trade-off. Gradual type systems have soundness and blame tracking as high priorities but restrict themselves to only supporting type system features that can be checked at tun-time. Some type system features that are hard to check at run-time, such as parametric polymorphism, are often missing from gradual type systems. On the other hand, Optional type systems often sacrifice soundness to make the type system less restrictive or to include conflicting features in the type system.

Soft and Success Typing

Soft Type systems are designed to infer types without the need for type annotation or other forms of human intervention. Because of this, these type systems must be very flexible. Even the most simple soft type systems found in papers include advanced type system features, such as recursive types and subtyping.

The main difference between traditional Soft Typing and Success Typing is the treatment of potentially-unsafe operations. Traditional Soft Typing issues warnings for the potentially-unsafe operations while Success Typing is optimistic sees these operations as potentially-safe. This choice will end up resulting in a Soundness vs Simplicity trade-off in the type system design.

In traditional type systems, well-typed programs never "go wrong" and potentially-unsafe programs are considered ill-typed. From this point of view, traditional Soft Typing would be considered to be a sound type system, while success typing systems would be seen as unsound.

However, soundness comes at a cost of complexity. In a soft-typing setting, the only way to get rid of a compiler warning is to rewrite the program to please the type system. It is not possible to just fall back to dynamic typing, as would be done in type-systems based around type annotations. Because of this, issuing warnings about potentially-unsafe operations since programmers are forced to perform "useless" rewrites to get rid of the warning. In traditional Soft Typing systems, the only way to minimize these false positives is to make type inference as precise as possible, to avoid classifying an operation as potentially unsafe. This added accuracy comes at a high complexity cost – traditional soft typing systems are notoriously hard to use [29] and the programmer needs to learn the nuances of type inference to be able to understand the error messages. Conversely, Success Typing systems avoid this problem in the first place, by never complaining about potentially-unsafe operations. This allows the type system to remain simple and with smaller, easier to understand inferred types.

8.3 Miscellaneous Opinions and Recommendations

Type system designers often need to balance conflicting desires such as soundness, expressiveness, and simplicity. Because of this, no single approach for typing dynamic languages is going to be superior to all the others and the ideal approach is going to depend on what properties of static and dynamic typing are most desirable to preserve. In this section we highlight some miscellaneous points that highlight positive aspects of different type systems that we analyzed.

Soundness is relative

Type-system soundness is usually defined along the lines of the motto "well-typed programs don't go wrong", which means that evaluation of well-typed programs never gets stuck or runs into undefined behavior. However, maybe for type systems for dynamic languages there should be more than one definition of soundness.

- The first definition is the traditional one: well-typed programs do not get stuck. All the type systems mentioned in this text are sound according to this definition, except for the type specifiers of Common LISP, which sacrifice soundness for performance.
- The second kind of soundness is whether type declarations written by the programmer are respected at runtime, which is the main difference between Gradual Typing and Optional Typing. If annotations are respected, then they can be used for optimization purposes. On the other hand, enforcing this level of correctness in an efficient manner is still an open problem.

Error messages should be actionable

An important factor for the usability of a type system is how easy it is to understand type-error messages and how easy it is to locate what part of the program needs to be fixed to make the error go away.

The largest evidence for this are the success-typing systems. The understandability of error messages was one of the factors why flow-based inference was more successful than unification-based alternatives. However, the only soft-typing system to really gain traction outside academia was the success-typing system. A success-typing system produces less error messages but the error messages they produce will almost always correspond to real bugs.

Explicit types are good for documentation and tool support

The large popularity of optionally typed languages relative to the other approaches discussed in this text suggests that type annotations are very useful, even if they are not providing many static guarantees. Just being able to document the types of APIs empowering IDEs with autocompletion is apparently very useful in practice, as can be seen by the popularity of Typescript.

Type inference has a niche in bug-finding

The biggest success case for type inference among the type systems in this text is how it enabled the Dialyzer tool to search for bugs on codebases with millions of lines of code, without any need for human intervention.

Except for the soft-typing systems, most of the type systems did not depend on type inferencing. In many of the type systems for dynamic languages, the parts of the program that have no type annotations are considered to be dynamically typed instead of making them be statically typed with an inferred type.

One possible explanation for this is that in statically-typed languages, type inference is fundamental for writing real programs, since no programmer would be bothered to use the programming language if it required extensive type annotations everywhere. On the other hand, in dynamic languages there is always the option of using dynamic typing when there are no annotations.

Extracting performance from type systems for dynamic languages is hard

One of the largest challenges for type systems for dynamic languages is being able to preserve as much of the intrinsic flexibility of the dynamic language as possible. However, if we ignore the gradually-typed systems, which are still an open area of research (as discussed in Section 6.4), then type systems in this text tend to sacrifice either soundness or performance. The Common Lisp type declarations allow undefined behavior and the Optional Typing systems give up on performance.

Soft typing does not really count for this trade-off between safety and performance. Soft typing is limited to types that can be automatically inferred and compilers for dynamic languages can already use type inference internally, without exposing it to the programmer, as it is done with Soft Typing.

Summing up

Combining the advantages of statically-typed and dynamically-typed programming is a worthy goal. However, it is not possible to simultaneously combine all the advantages of the two approaches in a single language. At a fundamental level there will always exist a trade-off between the expressiveness of the type system and its simplicity.

As we discussed in Section 1.1, there are multiple different reasons why static typing is appealing and different applications will prioritize them differently. We believe that if language designers and users are aware of which properties of static typing they prioritize more highly, then they will be able to make more informed choices.

9 Bibliography

[1] Jeff Williams. JSDoc documentation for the @type tag. User manual. URL http://usejsdoc.org/tags-type.html. Accessed in 17/03/2014.

[2] Neil Mitchell. *Hoogle Manual (Haskell Wiki)*, 2005. URL https://wiki.haskell.org/Hoogle.

[3] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, March 1998. ISSN 0018-9162. doi: 10.1109/2.660187. URL http://web.stanford.edu/~ouster/cgi-bin/papers/scripting.pdf.

[4] Douglas Crockford. RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). Technical report, IETF, 2006. URL http://tools.ietf.org/html/rfc4627.

[5] Douglas Crockford. Jsonobject class documentation, 2002. URL http://www.json.org/javadoc/org/json/JSONObject.html.

[6] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. doi: 10.1145/604174.604179. URL http://research.microsoft.com/en-us/um/people/simonpj/papers/hmap/index.htm.

[7] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1. URL http://www.cis.upenn.edu/~bcpierce/tapl.

[8] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. URL http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps.

[9] Gordon Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61(0):3 – 15, 2004. ISSN 1567-8326. doi: http://dx.doi.org/10.1016/j.jlap.2004.03.009. URL http://homepages.inf.ed.ac.uk/gdp/publications/Origins_SOS.pdf.

[10] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235 – 271, 1992. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90014-7. URL http://www.ccs.neu.edu/home/matthias/papers.html#tcs92-fh.

[11] Gilles Kahn. Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, pages 22–39, London, UK, UK, 1987. Springer-Verlag. ISBN 3-540-17219-X. doi: http://dx.doi.org/10.1007/BFb0039592.

[12] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, February 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2007.12.004. URL http://gallium.inria.fr/~xleroy/publi/coindsem-journal.pdf.

[13] Daniel Brown, Nuno Lopes, Felipe Pena, Thiago Pojda, and Maciek Sokolewicz. String conversion to numbers. PHP 5 Language Reference, august 2015. URL http://php.net/manual/en/language.types.string.php#language.types.string.conversion.

[14] Oleg Kiselyov and Chung-chieh Shan. Interpreting types as abstract values. Lecture notes from the Formosan Summer School on Logic, Language, and Computation, July 2008. URL http://okmij.org/ftp/Haskell/AlgorithmsH.html#teval.

[15] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions American Methematical Society*, 74:358–366, 1953. doi: 10.2307/1990888. URL http://www.ams.org/journals/tran/1953-074-02/S0002-9947-1953-0053041-6/home.html.

[16] Richard Statman. The typed λ-calculus is not elementary recursive. In *18th Annual Symposium on Foundations of Computer Science*, FOCS '77, pages 90–94, Oct 1977. doi: 10.1109/SFCS.1977.34.

[17] Robert Harper. *Practical Foundations for Programming Languages*, chapter 17, pages 149 – 150. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576. URL http://www.cs.cmu.edu/~rwh/plbook/book.pdf.

[18] John C. Reynolds. *Logical Foundations of Functional Programming*, chapter 5, pages 77–86. Addison-Wesley, 1990.

[19] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111 – 156, 1999. ISSN 0168-0072. doi: 10.1016/S0168-0072(98)00047-5.

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. ISSN 0022-0000. doi: 10.1016/0022-0000(78)90014-4.

[21] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: 10.1145/582153.582176.

[22] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference, pages 404–489. MIT Press, 2005.

[23] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 382–401. ACM, 1990. ISBN http://id.crossref.org/isbn/0897913434. doi: 10.1145/96709.96748.

[24] David McAllester. A logical algorithm for ML type inference. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, RTA'03, pages 436–451, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40254-3. doi: 10.1007/3-540-44881-0_31. URL `http://ttic.uchicago.edu/~dmcallester/rta03.ps`.

[25] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99404. URL `http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html`.

[26] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 205–215, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: 10.1145/141471.141542. URL `http://www.diku.dk/~henglein/bib/publications//henglein92c.html`.

[27] Konstantinos Sagonas. Experience from developing the dialyzer: A static analysis tool detecting defects in erlang applications. Presented at the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools (Bugs'05), June 2005. URL `http://user.it.uu.se/~kostis/Papers/bugs05.pdf`. This conference did not have formal proceedings.

[28] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 23–32. Association for Computing Machinery (ACM), 1996. ISBN http://id.crossref.org/isbn/0897917952. doi: 10.1145/231379.231387. URL `http://cs.brown.edu/~sk/Publications/Papers/Published/ffkwf-mrspidey/`.

[29] Matthias Felleisen. From soft scheme to typed scheme: Experiences from 20 years of script evolution, and some ideas on what works. 1st International Workshop on Script to Program Evolution (STOP 2009), July 2009. URL `http://ccs.neu.edu/home/matthias/Presentations/STOP/stop.pdf`. Invited talk.

[30] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and*

Implementation, PLDI '91, pages 278–292, New York, NY, USA, May 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113469. URL http://doi.acm.org/10.1145/113445.113469.

[31] Mike Fagan. *Soft Typing: an approach to type checking for dynamically typed languages*. PhD thesis, Rice University, April 1991. URL http://www.ccs.neu.edu/scheme/pubs/thesis-fagan.ps.gz.

[32] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 367–378, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199533. URL http://www.cs.ucla.edu/~palsberg/paper/toplas95-po.pdf.

[33] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997. ISSN 0164-0925. doi: 10.1145/239912.239917.

[34] Andrew K. Wright. *Practical Soft Typing*. PhD thesis, Rice University, 1994. URL http://www.cs.rice.edu/CS/PLT/Publications/Scheme/thesis-wright.ps.gz.

[35] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information Computation*, 93(1):1–15, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90050-C. URL http://www.ccs.neu.edu/home/wand/Bibliography.html.

[36] Didier Rémy. Type inference for records in a natural extension of ML. Technical report, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, May 1991. URL http://gallium.inria.fr/~remy/ftp/type-inf-records.pdf.

[37] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Inf. Comput.*, 118(1):128–141, April 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1058. URL http://www.cs.ucla.edu/~palsberg/paper/ic95-ps.pdf.

[38] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. URL http://ccs.neu.edu/home/shivers/papers/diss.pdf.

[39] Jan Midtgaard. Control-flow analysis of functional programs. Technical report, University of Aarhus, 2007. URL http://www.brics.dk/RS/07/18/index.html.

[40] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1 edition, 1999. doi: 10.1007/978-3-662-03811-6.

[41] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007. URL http://matt.might.net/papers/might2007diss.pdf.

[42] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions On Programing Languages And Systems*, 15(4): 575–631, September 1993. ISSN 0164-0925. doi: 10.1145/155183. 155231. URL http://lucacardelli.name/indexPapers.html#Subtyping%20recursive%20types.

[43] Jim Trevor and Jens Palsberg. Type inference in systems of recursive types with subtyping. Unpublished manuscript, January 1999. URL http://www.cs.ucla.edu/~palsberg/draft/jim-palsberg99.pdf.

[44] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, May 1997. URL https://users.soe.ucsc.edu/~cormac/papers/thesis.pdf.

[45] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, HASKELL '03, pages 62 – 71. ACM, 2003. ISBN http://id.crossref.org/isbn/1581137583. doi: 10.1145/871895.871902. URL http://www.open.ou.nl/bhr/HeliumCompiler.html.

[46] Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, University of Utrecht, 2005. URL http://dspace.library.uu.nl/handle/1874/7297.

[47] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178. Association for Computing Machinery (ACM), July 2006. ISBN http://id.crossref.org/isbn/1595933883. doi: 10.1145/1140335.1140356. URL http://it.uu.se/research/group/hipe/papers/succ_types.pdf.

[48] Tamás Nagy and Anikó Nagyné Víg. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 21–28, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: 10.1145/1411273.1411277.

[49] Dialyzer release notes. Erlang/OTP official documentation. Available at http://www.erlang.org/doc/apps/dialyzer/notes.html. URL http://www.erlang.org/doc/apps/dialyzer/notes.html.

[50] Fred Hébert. *Learn You Some Erlang for Great Good!* No Starch Press, 2013. ISBN 978-1-59327-435-1. URL http://learnyousomeerlang.com/.

[51] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems*, APLAS '04, pages 86–101, 2004. ISBN http://id.crossref.org/isbn/978-3-540-30477-7. doi: 10.1007/978-3-540-30477-7_7. URL http://user.it.uu.se/~kostis/Papers/war_story.pdf.

[52] Guy L. Steele, Jr. An overview of common lisp. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 98–107, New York, NY, USA, 1982. ACM. ISBN 0-89791-082-6. doi: 10.1145/800068.802140.

[53] David A Moon. *MACLISP reference manual*. Massachusetts Institute of Technology, April 1974. URL http://www.softwarepreservation.org/projects/LISP/MIT/. The "Moonual".

[54] Paul Graham. *ANSI Common LISP*. Apt, Alan R., 1996. ISBN 0-13-370875-6. URL http://www.paulgraham.com/acl.html.

[55] Steel bank common lisp. URL http://www.sbcl.org/.

[56] J. S. Foster. Safe programming in dynamic languages. NSF Workshop on Programming with Big Data, Jan 2013. URL http://janvitek.github.io/events/PBD13/slides/JeffFoster.pdf. Talk.

[57] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. *Scheme and Functional Programming Workshop*, 6:81–92, 2006. URL http://www.cs.colorado.edu/~siek/pubs/pubs/2006/siek06:_gradual.pdf.

[58] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*, ECOOP '07, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73588-5. doi: 10.1007/978-3-540-73589-2_2. URL http://ece.colorado.edu/~siek/gradual-obj.pdf.

[59] 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. ISBN 978-1-4503-1895-2. doi: 10.1145/2661103.2661112.

[60] Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 27–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508171. URL http://pleiad.dcc.uchile.cl/papers/2013/allendeAl-dls2013.pdf.

[61] Jeremy G Siek, Michael M Vitousek, and Shashank Bharadwaj. Gradual typing for mutable objects. Unpublished manuscript, 2013. URL http://ece-www.colorado.edu/~siek/gtmo.pdf.

[62] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *28th European Conference on Object-Oriented Programming*, ECOOP '14, pages 257–281. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-44202-9_11. URL https://users.soe.ucsc.edu/~abadi/Papers/FTS-submitted.pdf.

[63] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661101. URL `http://wphomes.soic.indiana.edu/jsiek/`.

[64] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. 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 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_1. URL `http://homepages.inf.ed.ac.uk/wadler/topics/blame.html#blame-esop`.

[65] Sam Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, January 2010. URL `http://www.ccs.neu.edu/racket/pubs/dissertation-tobin-hochstadt.pdf`.

[66] Philip Wadler. A complement to blame. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 309–320, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.309. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5033`.

[67] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages*, SNAPL '2015, pages 274–293, Asilomar, California, USA, May 2015. doi: 10.4230/LIPIcs.SNAPL.2015.274. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5031/pdf/21.pdf`.

[68] RobertBruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, ECOOP '04, 2004. doi: 10.1007/978-3-540-24851-4_17. URL `http://www.ccs.neu.edu/home/matthias/papers.html#ecoop2004-fff`.

[69] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 201–214, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926409. URL `http://homepages.inf.ed.ac.uk/wadler/topics/blame.html#blame-for-all`.

[70] 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. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328486. URL http://arxiv.org/abs/1106.2575.

[71] Gilad Bracha. Pluggable type systems. OOPSLA04 Workshop on Revival of Dynamic Languages, 2004. URL http://bracha.org/pluggableTypesPosition.pdf.

[72] 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. ISBN 0-89791-587-9. doi: 10.1145/165854.165893. URL http://www.bracha.org/oopsla93.ps.

[73] André Murbach Maidl. *Typed Lua: An Optional Type System for Lua*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, April 2015. URL https://github.com/andremm/typedlua.

[74] Google. Dart programming language. https://www.dartlang.org/, 2011. URL https://www.dartlang.org/. Official Website.

[75] Source code repository for the angular framework. Available at https://github.com/angular/angular. URL https://github.com/angular/angular.

[76] Jonathan Turner. Using typescript in visual studio code. Blog post available at http://blogs.msdn.com/b/typescript/archive/2015/04/30/using-typescript-in-visual-studio-code.aspx, 2015 April. URL http://blogs.msdn.com/b/typescript/archive/2015/04/30/using-typescript-in-visual-studio-code.aspx.

[77] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP'15)*, ECOOP '15, pages 999–1023, 2015. URL http://ccs.neu.edu/racket/pubs/ecoop2015-takikawa-et-al.pdf.

[78] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 306–317, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182495. URL http://www.cs.cmu.edu/afs/cs/user/nch/www/sba.html.