12-1-2022

Jiapi: A Type Checker Generator for Statically Typed Languages

Benjamin Cisneros Merino

JIAPI: A TYPE CHECKER GENERATOR FOR STATICALLY TYPED LANGUAGES

By

Benjamin Cisneros Merino

Master of Science – Computer Science
University of Nevada, Las Vegas
2019

Bachelor of Science – Computer Science
University of Nevada, Las Vegas
2017

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2022

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

December 5, 2022

This dissertation prepared by

Benjamin Cisneros Merino

entitled

Jiapi: A Type Checker Generator for Statically Typed Languages

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy - Computer Science
Department of Computer Science

Jan Pedersen, Ph.D.
Examination Committee Chair

Kazem Taghva, Ph.D.
Examination Committee Member

Fatma Nasoz, Ph.D.
Examination Committee Member

Andreas Stefik, Ph.D.
Examination Committee Member

John Minor, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Alyssa Crittenden, Ph.D.
Vice Provost for Graduate Education & Dean of the Graduate College

Abstract

Type systems are a key characteristic in the context of the study of programming languages. They frequently offer a simple, intuitive way of expressing and testing the fundamental structure of programs. This is especially true when types are used to provide formal, machine-checked documentation for an implementation. For example, the absence of type errors in code prior to execution is what type systems for static programming languages are designed to assure, and in the literature, type systems that satisfy this requirement are referred to as *sound type systems*. Types also define module interfaces, making them essential for achieving and maintaining consistency in large software systems. On these accounts, type systems can enable early detection of program errors and vastly improve the process of understanding unfamiliar code. However, even for verification professionals, creating mechanized type soundness proofs using the tools and procedures readily available today is a difficult undertaking.

Given this result, it is only logical to wonder if we can capture and check more of the program structure, so the next two questions concern *type checking*. Once a type system has been specified, how can we implement a type checker for the language? Another common question concerns the operational semantics, that is, once the type system is specified, how can we take advantage of this specification to obtain a type checker for the language? In each of these cases, type systems are often too complex to allow for a practical and reasonable approach to specifying many common language features in a natural fashion; therefore, the crux here is: **What is the bare minimum set of constructs required to write a type checker specification for any domain-specific language (DSL)?** We believe that sets (including tests for memberships, subsets, size, and other properties) as well as first-order logic and an expression grammar can be used to do this. In addition, we will need the ability

to apply some form of filter on sets, as well as possibly a reduction and map.

The primary goal of this thesis is to raise the degree of automation for type checking, so we present *Jiapi*, an automatic type checker generation tool. The tool generates a type checker based on a description of a language's type system expressed in set notation and first-order logic, allowing types to be interpreted as propositions and values to be interpreted as proofs of these propositions, which we can then reduce to a small number of easily-reviewable predicates. Consequently, we can have a completely spelled-out model of a language's type system that is free of inconsistencies or ambiguities, and we can mathematically demonstrate features of the language and programs written in it. We use two languages as case studies: a somewhat large and more general, and a more domain-specific. The first one is a true subset of Java (an object-oriented language) called **Espresso**, and the second one is a pedagogical language like C/C++ called **C-Minor**. Also, there is a declarative type system description extension for the first language, and parts of the specification for the second one. Finally, our approach is evaluated to demonstrate that the generated type checker can check automatically the full functional correctness of an Espresso and C Minor program.

Acknowledgements

"There are a number of people I not only want to thank but also express my gratitude to. Firstly, I would like to express my deep gratitude to my Ph.D. advisor Dr. Jan 'Matt' Pedersen. My interest in interpreters and compilers began with a Spring course taught by him at the University of Nevada Las Vegas. His experience and approach to the field have shaped my views on various programming languages considerably. I would also like to thank all the members of my committee and for their valuable input.

Last but not the least, I would like to thank my family. I feel immense gratitude for their constant support and love during the entire process of writing this thesis as I sat endless hours in front of my computer."

Benjamin Cisneros Merino

University of Nevada, Las Vegas

December 2022

To my Daughter

You had me wait 9 long months, and then you arrived in a swirl of drama just like a celebrity diva... But it does not matter because I will always be your biggest fan.

To her Mommy

For sticking with me through this whole roller coaster, for tolerating my child behavior, for every now and then giving a courtesy laugh at my lousy dad jokes, and for helping me with the diapers... thank you.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Overview

As statically typed languages become increasingly sophisticated, difficult to learn or even understand [1, 2] (e.g., such as C++), we discover new reasons to *reject* programs we do not want to compile: Names that are not in scope, new definitions that hide prior definitions, functions that are called with the incorrect number of arguments, and employing functions or operators with the incorrect types of arguments. To this list, we can add another issue: The number of features a language contains, or how many approaches there are to achieve the same goal, is a measure of its complexity. To prevent some of these drawbacks, it makes logical sense to try to create a *type system* that can determine whether or not a particular input program is internally consistent with its data usage at compilation time.

In its most basic form, a type system enable us to specify the purpose of a program in the form of a *type*. For example, the purpose of finding the sum of elements in an array is to take an array as input and produce the sum of all of its elements as output. This is a (fairly) basic approximation of what it takes to write a summation program, but it provides certain assurances. If the array's type is known at compile-time (before the program executes), the compiler can deduce the types of all of its elements as well. If the array is declared as integer, then the compiler concludes that adding each of its elements results in an integer type. Thus, the more specific we make the types, the more we provide a way of defining type-constraints which say what type our variables have.

A **type system** is therefore an important component of a compiler's semantic analysis phase. It is built into most programming languages to detect *type errors* in programs, such as when a value is used in a way that is inconsistent with its definition, or using the result of an expression of one type in a context that expects data of a different and, perhaps, incompatible type [3, 4, 5, 6, 7]. This type of verification model − known as **type checking** − can be done statically (at compile-time) or dynamically (during run-time). It follows that type checking has proven to be quite effective in preventing − and catching − a wide range of programming errors by ensuring that valid operations are invoked on variables and expressions, enforcing intended interpretation of values, and providing a concise formalization of the semantic checking rules.

Other means of testing the correctness of a program, such as ensuring that data has the expected type without cluttering code with dynamic checks or having run-time errors possibly from logical errors, examining the boundaries of an array [8], avoiding improper pointer traversals, preventing unintentional aliasing [9], and examining control flow, are also included in the compiler's semantic analysis phase. Naturally, depending on the design of the language, some of these checks can be resolved at compile-time, while others may have to wait until run-time. It is clear, though, that the semantic-checking process may include many more stages − or passes − depending on the language for which the compiler is being written.

My thesis In this dissertation, the focus of semantic analysis is **type checking**. Figure 2 is a mere adaptation and variation of [5, Figure 1.6] and [3, Figure 1.7] that demonstrates how the implementation of a generated type checker can change the structure of a compiler. Here, a generated type checker is a program that takes a description of a programming language's type system as input and generates a type checker for use in the semantic analysis phase of the language's compiler. Although it is common knowledge that a type checker, just like a scanner, parser, and code generator, requires careful specification and implementation by the compiler writer, unlike these other components, a type checker does *not* have access to specialized tools. We anticipate that by using our approach, the parser and type checker will be closely integrated, making the development of new languages easier. This means that

a type checker can be written abstractly, without having to worry about implementation details, allowing language designers and everyday compiler writers to focus on the essentials. That is, rather than adhering to a specific grammar, a higher-level of abstraction can be used to make it easier to check for errors. To illustrate this point, consider these questions: How can we specify the range of an array? How many operations can we do on an array? etc. Clearly, it should be feasible to construct an abstract interpretation of an array by considering its terms as separate objects and abstracting away the actual values such objects can have by focusing just on the kind (types) to which they belong to.

In this dissertation, we also describe a specific implementation of the above ideas using an object-oriented language, in which the evidence of type safety is presented in the form of a proof that the type checker validates to satisfy a safety specification. As we shall see later, a type system consists of a set of typing rules for a programming language's many syntactic constructs. In most cases, the typing rules take the form of an implication; for example, the addition typing rule may be $(x : int \wedge y : int) \Rightarrow (x + y) : int$, which means that if the type system can type both x and y, then it can also type the expression $x + y$. Typing rules are usually described as inference rules in the literature on type systems, with all premises put above the bar and the conclusion stated below the bar. A side-by-side comparison of our methodology (left) and the traditional method of specifying a type system (right) is shown in Fig. 1. (We will elaborate more on how to represents things like "is an integer".)

$$\forall x, y : y \text{ is an integer} \wedge x \text{ is an integer} \Rightarrow (x + y) \text{ is an integer}$$

$$\frac{\overline{\Gamma \vdash x : \texttt{int}} \quad \overline{\Gamma \vdash y : \texttt{int}}}{\Gamma \vdash (x + y) : \texttt{int}}\textbf{add}$$

Figure 1. Side by side comparison.

It is important to note that the proof and the specification are both written in a mathematical logic, which is an extension of first-order predicate logic in this case. We also employ small-step operational semantics, which involves reducing expressions in small steps until they achieve a normal form. For example, an expression 2×4 evaluates further to the expression 8 via an evaluation rule $e \rightarrow e'$. The rest of this dissertation's ideas are techniques to get around the inherent complexity of static analyses and the indeterminacy of many useful language features.

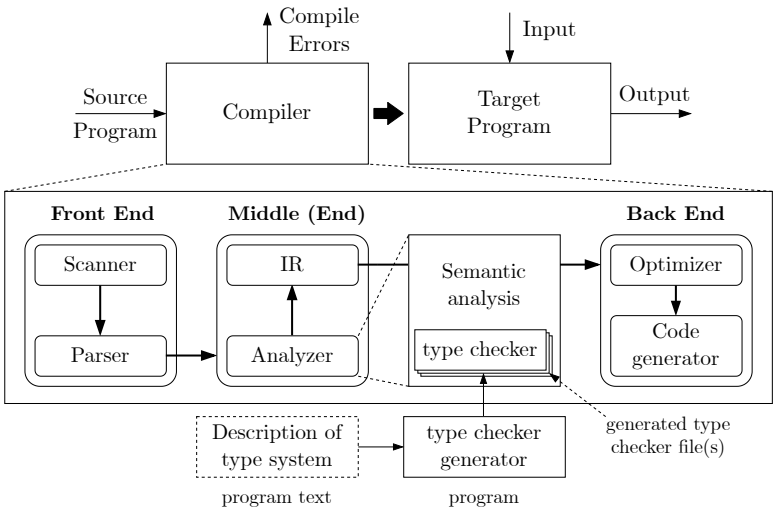


Figure 2. The generated type checker.

Remark 1 (Type Checking): For some languages, the type checker may only need to verify that the programmer's type annotations are consistent. For instance, a type checker can alert the programmer at an early stage (during compilation) if a program contains *sever errors*, such as applying a function defined for *double* to a *string*. For others, however, the type checker may need to conduct type inference and, in some instances, reconstruct the type of an expression[1]. Fortunately, the type checker in most languages is somewhere in the middle of these two ends, inferring at least the "obvious" types, such as result of expressions: The result of a type expression is either reconstructed or matched against some type of signature provided by the programmer. Regardless of the component's real function, in this thesis, we will refer to it as a type checker. □

1.2 Types and Type Systems

In general, types and type systems are indispensable in programming languages, as they provide abstractions, documentation, helpful invariants to programmers, and can also be used to instruct a programmer on how to construct correct programs. Even though the semantics of a programming language defines **types** as subsets of the computed value domain, its **type system** allows programmers to make verifiable statements − or *assertions* − that a compiler

[1] In Espresso [10], as well as in ProcessJ [11], for example, the real *baseType* of a two-dimensional array changes to be an array with one dimension removed (see Section 4.2 on page 59).

can check automatically, allowing errors to be detected at compile-time rather than at run-time; however, these verifiable statements are handled differently in different languages. For example, some languages (such as C/C++ [12]) have **weak type systems**, making critical errors possible to emerge if not careful, whereas others (such as occam [13, 14, 15] and occam-π [16, 17, 18, 19, 20]) have very **strong type systems**, making writing useful programs more challenging if at all even possible to compile (e.g., [21, Section 5.1.1][2], [22, see occam-π's documentation]).

Despite the fact that different type systems are designed to eliminate different classes of programming errors [23], such as *trapped errors*, *untrapped errors*, or *forbidden errors* (see Section 3.4 on page 39), the majority of them strive to eliminate **type errors**. In this way, by employing types and a well-designed type system, a language can often guarantee that desirable behaviors are preserved and undesirable behaviors are avoided. This makes a program significantly safer as it ensures data structure integrity and type-correct coupling of program components (i.e., there is a level of assurance that the program will not carry out any undefined or prohibited operations).

Unfortunately, there are no obvious and unambiguous guidelines − except for the standard language evaluation criteria of [24, Section 1.3] and selected criteria of [25, Section 1.2] − when it comes to language design. However, when designing the type system of a language, a number of questions often arise; the bullet points that follow address some of these questions:

- **Soundness vs. completeness:** How many meaningful programs should be accepted? Should all invalid programs and some valid ones be rejected? Or, should all valid programs but also some invalid ones be accepted? For example, because of the object reference downcasting in a class hierarchy, some invalid programs are admitted in a typical object-oriented language (such as Java, C++, and C#) when, perhaps, they should not [5]: If A is a superclass of B and C, a reference to an object of class B or C may be stored in a variable of type A. A conversion from type A to either type B or C requires a run-time check to ensure that the run-time value is actually

[2] Non-aliasing in occam/occam-π prevents the implementation of several useful data structures like linked lists, circular lists, trees, etc.

an instance of the correct class; this is something that cannot always be checked at compile-time.

- **Decidability:** Is it possible to determine whether a given expression is well-typed? In a statically typed language, this is critical because the language's compiler must be able to determine whether a given input program is meaningful and then either run or reject it [5]. Thus, is it possible to implement a type checker that decides if a program is well-typed? Unfortunately, the answer is *no*. For example, C# [26] has nominal subtyping, that is, a type T_1 is a subtype of another type T_2 if and only if it is explicitly specified as such[3]. Additionally, C# has generic types, and covariance and contravariance of generic interfaces. This means that for a parameterzied − or generic − class C<T>, a contravariant class will allow the use of a more generic type, whereas a covariant class will allow the use of a more specific type (than what was initially specified). These three things are sufficient to make C#'s type system − and any language that shares similar characteristics (see also [28, 29, 30, 31, 32, 33, 34] for various approaches) − undecidable.

- **Effectiveness:** Aside from the theoretical aspect of decidability, there is this singular question of type checking efficacy that must also be taken into consideration. For example, is it possible to make type checking decidable in object-oriented languages? [35, 36, 37, 38], and [39] describe what restrictions can be put on a nominal subtyping system − for example, such as C#'s − to make it decidable. In particular, [35] focuses on the combination of three main features: Subtyping is decidable

  (*i*) if the class table does not use contravariance. (A class table is a collection of class declarations of the form C<$\overline{X}$> <:: $T_1 ... T_n$, where C is the class name and $T_i s$ may refer to the parameter variables $\overline{X}$ [35].)

  (*ii*) if the class table is not *expansive*, that is, there is no cycle in any inheritance hierarchy. (An expansive class table is defined by graphs having a cycle with at least one expansive edge. For example, either $C\#i \xrightarrow{1} C\#i$ − a one cycle,

[3] Note that the subtype relation in nominal type systems is between type names, and subtype relationships are explicitly defined [27].

or $C\#i \xrightarrow{1} D\#j \rightarrow^{+} C\#i$, where $C\#i$ is the i^{th} formal type parameter in the definition of the class C (i.e., $C\texttt{<}\overline{X}\texttt{>}$) [35].)

(*iii*) if multiple instantiation inheritance is not used and all expansive-recursive type parameters are invariant and linear.

However, the situation for many object-oriented languages remains the same: We pay the price for repetitive run-time checks, a lack of guarantees that hidden bugs are no longer lurking in our code, a lack of unambiguous attribution of how implausible data came to be used in the way that it was, and due to a lack of knowledge of the concepts behind object-orientation [40, 41].

- **Accuracy:** To what extent can the properties of programs be described? For example, an integer division, such as 7/3, does not return a fraction; it returns a truncated integer instead. This is an example of a situation where a run-time error (since a fraction cannot be represented as an integer) often results in the incorrect answer. Another example is integer overflow. What happens if the result of a calculation is too large (positive or negative) to fit within a fixed range? The computation discreetly wraps around (i.e., it overflows) and produces an integer that is within the acceptable range, but not the correct value [42]. As a last example, floating point and double types in Java have several non-real-number values, such as `NaN` (or "Not a Number"), `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY` [43, 44]. Thus, operations that should create run-time errors instead produce one of these unusual values. However, when the value being converted is something else, such as a `String` that does not represent a number, a conversion will result in `NaN`.

- **Compactness:** What is the minimum amount of type information that the programmer must provide? And how much of that information should be organized? If there is a standard means of recreating the missing information, then allowing the programmer to omit particular types is favorable. For example, type inference (such as the `auto` keyword) and move semantics (such as structured binding or decomposition declarations) in C++ [45]. For more general type systems, on the other hand, decidability

can commonly be retrieved by requiring a large number of type annotations [46]. For example, the `@TypeChecked` annotation in Groovy allows for a static type checker to be invoked at compilation-time [47]. `@TypeChecked` also enables the use of *abstract syntax trees* (ASTs) for compile-time metaprogramming [48, Section 2] and instructs the compiler to omit certain type checks via type checking extensions. The checking process for a Groovy class, thus, includes verifying that fields and properties exist at compile-time. Annotations can also be written in additional locations [49, 50], such as generic type parameters (e.g., `List<@NonNull Object>`), thanks to Java 8's type annotation syntax.

Realistically, **soundness**, **completeness**, and **decidability**, are in constant disagreement: If we can catch all type errors at compile-time, the language is said to be strongly typed and, therefore, the type system must be sound and decidable; but if we cannot, the type system is complete but not sound and, therefore, not decidable. Notably, it is reasonable to expect a type system to be difficult to get right, and getting it wrong might result in a slew of inconsistencies. For example, because of the object reference downcasting, we know that Java is not strongly typed. Hence, it is possible to write Java programs that are invalid with respect to the type system [5] (i.e., while the Java type system is complete, it is not *sound* and not *decidable*). Fortunately, an endless number of small and gradual adjustments can be made to a type system (e.g., Java's [51, 52, 53, 54]) to make it stronger [55, 56, 57, 58].

Definition 1 (Sound and Completeness): If all invalid programs are rejected, a type system is said to be sound, and if all valid programs are accepted, it is said to be complete. If a type system is sound but no complete, it will reject both invalid and valid programs (we have false negatives). However, if a type system is complete but not sound, it will accept all valid programs as well as some that are not (we have false positives). Furthermore, a type system is decidable if a type checker that decides if a program is well-typed can be implemented [5]. □

1.3 Describing a Type System

The addition of type systems to statically typed languages is motivated by more than just safety. Another motivation is to limit − and possibly eliminate − programs that contain constructs that are difficult to compile correctly or even efficiently [59]; in reality, useful type systems have a certain degree of freedom in expressing types and their relationships, which requires that the type system be applied to the operators and operands of the language [5], as well as establishing a mechanism for associating types with specific language constructs.

In spite of that, there is a broad spectrum of type systems, each with its own goals, emphasis, and application areas, but they all share a similar inclination for expressing implications between *typing judgments*, as seen in the use of deduction − or inference − rules in their description [4, 60, 61]. For simplicity, consider the following formalism for describing constants, variables, functions, and function applications in the simply-typed λ-calculus (or typed λ-calculus for short), as described in [4, 56, 60, 62].

$$\frac{}{\vdash e : \tau}(1) \quad \frac{(e : \tau) \in \Gamma}{\Gamma \vdash e : \tau}(2) \quad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\, e_2) : \tau'}(3) \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau . e) : \tau \to \tau'}(4)$$

The rules for deriving typing judgments are as follows. A typing judgment has the form $e : \tau$ (an assertion), which means that if e is an expression, it produces a value described by τ when its evaluation terminates. This also requires that e be *well-typed* before predicting e's ultimate value.

The first rule simply says that a constant value e has type τ (where τ is a meta-variable). However, when we need to type check the type of a variable, we require a new typing rule that allows us to leverage the typing context (i.e., we need the context of a type environment like a symbol table). Thus, the second rule stipulates that any expression e must have at least one type τ under the assumption that free variables in e have types specified in Γ if and only if $e : \tau \in \Gamma$ (where Γ is the type environment)[4]. The third rule says that if we have a function of type $\tau \to \tau'$ and an argument e_2 of type τ, then applying e_1 to e_2 will produce an expression of type τ'. In a similar manner, the last rule states that $\lambda x : \tau . e$ is an expression denoting a function of type $\tau \to \tau'$, whose formal parameter x has type τ,

[4] A type environment is a partial function that assigns types to variables in this context, and it is usually denoted with Γ where $\Gamma \in Env : Var \rightharpoonup Type$.

and the function body e has type τ'. The notation $\Gamma, x : \tau$ (or $\Gamma[x \mapsto \tau]$) is an extension of the typing context Γ that includes a new mapping from x to τ. Thus, if Γ already has a mapping for x, this new mapping takes precedence over it.

For completeness, we will illustrate these key ideas and define a set of inference rules that allow us to make judgments about expressions in typed λ-calculus for the Espresso language. Part of the formalized syntax of Espresso is defined inductively in Figure 3. Although we will demonstrate how expressions and statements in a (very extensive!) grammar can be used to describe any type of expression, we will largely employ shortcuts to define parts of Espresso's syntax.

$\tau \in$ primitives	$=$	**byte** | **short** | **char** | **int** | **long**	types
	|	**float** | **double** | **boolean** | **void**	
	|	$\tau \to \tau'$	function
$e \in$ expressions	$=$	v	variables
	|	**true** | **false**	boolean values
	|	$e_1 \ominus e_2$	binary operations
	|	...	
$s \in$ statements	$=$	**if** `(` e `)` `{` s_1 `}` **else** `{` s_2 `}`	if statement
	|	...	

$$\ominus \in \{\texttt{*}, \texttt{\%}, \texttt{/}, \texttt{+}, \texttt{-}, \texttt{<<}, \texttt{>>}, \texttt{<}, \leq, \texttt{>}, \geq, ...\}$$

Figure 3. Part of the formalized syntax of Espresso.

Espresso is a language that is quite similar to Java but without exceptions, significantly simpler imports, and without generics [10] (basically, Java 1.0). It is a fairly complex programming language with constructs for arithmetical operations, lexical binding of variables, scope, classes, functions, assignments and side effects, simple control constructs (including sequential, selection, and repetition), and dynamic dispatch. Hence, its syntax contains a number of properties and features that can be handled by the analysis described above. For example, a simple arithmetical rule for adding two integers, and an integer rule (an axiom) that says that n always has the type `int`, can be defined as follows:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\textbf{add} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\textbf{int}$$

The rule `add` reads aloud as "if e_1 has type `int` and e_2 has type `int`, then so does $e_1 + e_2$"; specifically, the colon functions as the phrase "has type", whereas the bar acts as

an *if-then* clause. For example, to deduce that the expression $7 + (3 + 2)$ is of type `int`, we simply need to link several uses of the rule `int` into a *derivation tree*, with the root of the tree being the judgment we want to prove, to derive that such expression has type `int`. Note that the leaves of the tree (dashed rectangles), as shown in the diagram below (see Figure 4), are all *axioms*. We began with a conclusion rather than ground truths since we were looking for a derivation that proved the proposition (i.e., the root).

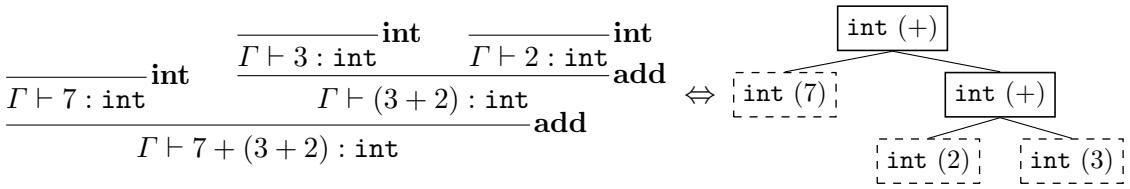


Figure 4. Derivation using type rules.

A type checker could then implement the rule `add` in three steps:

1. Type check e_1 : `int`.

2. Type check e_2 : `int`.

3. The type of the entire expression is `int`.

It is important to note that the type of each expression is checked in a predetermined order (commonly in the form of a *visit* [63, 64]), as the type of e_1 and e_2 must be known before addition can be computed, which was not the case in the above derivation tree. Furthermore, since a type checker is basically a theorem-prover that is formally validating the type of an expression, we can think of Γ as a "proof context". That is, in order to type check expressions with variables, we need to introduce a typing context − a symbol table of some sort − that maps variables to their types. For example, a variable can have *any* of the types available in the language. In Espresso, the type is determined by the **declaration** of the variable. In inference rules, the variables are collected to a context, denoted by Γ; however, for the compiler, the context is a symbol table of (variable, type) pairs. This symbol table is created whenever a language construct opens a new scope so that names can be associated with their declarations/definitions [5], facilitating type checking of variables in expressions.

Naturally, we say that a correct program is also one that never refers to an *undefined variable* (i.e., a variable that is not accessible or was not declared in the program). Many compilers do a check to determine if variables and other named entities have been declared or defined before they are used. These entities cannot be referred to in locations where it would be considered illegal or out of scope; therefore, symbol tables have an important part in type checking. A rule for checking whether a variable is defined in some context can be defined as follows:

$$\frac{}{\Gamma \vdash v : \tau}\textbf{var},\ \Gamma(v) = \tau$$

The rule `var`[5] states that variable v has whatever type the context assigns to it (i.e., $\tau \in \{byte, short, int, long, float, double, boolean, void\}$), which means that Γ must have an association for v in order for $\Gamma \vdash v : \tau$ to hold. In Espresso, we can formulate name resolution as a two-stage problem when dealing with defining and resolving symbols, such as names of variables, classes, and methods [5]. The first stage is to locate and insert all named entities that can be forward referred into the proper symbol tables. Meanwhile, the second stage involves resolving the use of all names, including variables names, parameter names, and method names. The symbol table is essentially a map between each variable's name and the symbol structure that describes it.

Following the example of the first arithmetical rule (i.e., `add`), it is natural that most of Espresso's operations for primitive types follow a similar pattern; for example, we could create a conversion rule that would change an *integer* to a *long*, *double*, or other type of number and use the transformed value as such. While this may appear to be a simple mathematical concept (given that integers are a subset of doubles), integers and doubles have completely different binary representations and sets of instructions. As a result, type conversions are normally handled by the compiler using a specific instruction. Consider the typing rule below (i.e., `max`):

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 + e_2 : \texttt{max}(\tau, \tau')}\textbf{convert},\ \textbf{if}\ \tau \in \{int, long, double\}$$

If we were to assume no loss of information and the following ordering:

$$int <_{\tau} long <_{\tau} double$$

[5] This rule is equivalent to rule (2) in Section 1.3 on page 9.

then, the type of $e_1 + e_2$ for $\tau = int$ and $\tau' = double$ is

$$\lceil int, double \rceil_{\boldsymbol{\tau}} = double$$

Thus, 2 + 2.4 gives the result 4.4 because *double* is the maximum. (Note that the $<_{\boldsymbol{\tau}}$ operator is introduced in Section 4.4.4 on page 75.)

At the same time, some of Espresso's types are more of a run-time notion (a term we use to refer to instances that represent an implementation of a type − a class) and, as such, can be manipulated at run-time, rendering compile-time checking impossible. In Section 4.2 on page 58, we will look at the primitive types, classes and inheritance, instance variables and methods, interfaces, shadowing of instance variables, and dynamic method binding in the Espresso language.

1.4 Proposed Approach

The problem with sophisticated type systems is that they force us to spend a lot of our development time (universally) thinking about type hierarchies and their interactions rather than actual reasoning (in a more localized way), oftentimes involving: Naming and organizing useful concepts (as seen earlier), providing information − to the compiler or programmer − about data manipulated by a program, and ensuring that the run-time behavior of programs meet certain criteria. This failure appears to be primarily attributable to the fact that type checkers are written in a pre-existing formalism rather than a type-specific syntax. For this reason, we suggest that a different kind of abstraction be created: One that is more simple and practical (though, perhaps, not as precise), and more "approachable" for the (average) compiler writer.

We present *Jiapi*, an automatic type checker generator tool that uses *first-order logic* and *set notation*, along with an expression grammar for describing the type system of any domain-specific language (DSL). To define a type system, *Jiapi* generalizes the concepts of atomic and constructed types and combines them with a controlled form of a meta-language. The intent is to automate code generation for semantic checking tasks that can be specified and inferred based on types alone. For the sake of clarity, this thesis focuses on the type safety of **Espresso** as an example, a large but representative Java subset. Correspondingly, the

abstract syntax, type system, and type constraints guaranteed by Espresso's type analysis are formally defined. The thesis also provides a formal specification of the type system rules, which a Espresso program must follow in order to be accepted.

Finally, what sets *Jiapi* apart from other approaches is that we want to create a formal verification model that can be explained as a simple extension of type checking verification, based on simple concepts that are used to describe a language construct (such as types, functions, operators, and so on), and for which we can specify the required checks − largely − in terms of **type predicates** (of the form $\mathcal{P}_?(x)$), **type equality** (the $=_\tau$ operator), **type equivalence** (the $\sim_\tau$ operator), and **assignment compatibility** (the $:=_\tau$ operator). In fact, undergraduate and graduate students in [65, a compiler construction course at UNLV] will be customers of this work, both in the classroom and at home, where they have already been using different verification approaches to improve the quality of their compilers.

1.5 Scope of Type Systems

As we wrap up this chapter and begin our analysis of the problem that our research attempts to address (in Chapter 2 on page 17), it is noteworthy to mention that type systems are far too vast a topic to include all proposed systems in a single framework. Therefore, we will briefly summarize the common ways type systems are specified and perform many semantic checking tasks in accordance with the literature.

Type systems are motivated by a variety of factors, including mathematical reasoning [66, 67, 68, 69, 70] and the need to differentiate data types during compilation [3, 5, 71]. They can be described in a variety of ways, including natural language [72, 73, 74], deduction systems [75, 76, 77], algorithms [78, 79, 80], and constraint systems [81, 82, 83]. Although they cover everything from checking type annotation correctness to reconstructing the type of every expression and function in a program [84, 85], they often operate by establishing and enforcing constrains. For example, some constrain the argument type of a function to be a subtype of type, such as $t \to int \backslash \{t \leq int\}$, where t is constrained to be a subtype of int, while others allow recursive constrains, such as $t \leq t \to int$ [83]. There are several type systems that add constraints to the Hindley–Miller system [86]; for instance, record

systems [87], overloading [88, 89, 90], and systems that permit subtyping[6] [93, 94] are just a few examples.

Compiler optimizations, static analysis, software design approaches, and safety considerations on executable code can all be found somewhere in the middle [3, 5]. For example, it is a static type system if the type system is present in the language and used at compile-time with the goal of being able to issue compile-time errors. Otherwise, it is a dynamic type system that prohibits illegal program states by stopping the program in the middle of its execution rather than preventing it from starting at all. In contrast, providing both static and dynamic characteristics [95] is a fairly frequent type system feature. Hybrid type checking combines the two approaches with the purpose of enforcing precise interface requirements when static analysis is possible and dynamic checks are required [96, 97]. This method is useful in dependent type systems where types are parameterized by expression run-time values[7] (e.g., [100] and [101]).

Even this rudimentary summary suggests that the scope of a type checker generator should be (and is) limited. For an analysis, known parser generators do **not** handle every imaginable language syntax, but they do provide a framework that can be easily customized to a specific class of syntax structures prevalent in computer languages. On account of this, we will start with the simply-typed lambda calculus while developing the type checker for Espresso, gradually adding language components to make the design self-contained and well-defined. Then, we will retrace these steps with the resulting system and re-write the implementation in *Jiapi*.

1.6 Road Map

The rest of the thesis is organized as follows. The first chapter (which you are currently reading) serves as an overview of the rest of the thesis. After reviewing the essential background on types and type systems, Chapter 2 begins with a general exposition of the problem, which

[6] John C. Mitchell was the first to propose that subtyping constraints be included in typing decisions judgments [91, 92]. Only coercions between type variables were allowed in his constraint sets, which were atomic.

[7] The separation between types and terms is reduced in dependently typed languages. They allow types to rely on a term of a different type, increasing the expressiveness of a programmer's code and lowering the computational cost while lowering the risk of type errors [98, 99].

later explains our central argument; it then goes on to explain why statically-typed languages are preferred, and ultimately, it describes our motivations and contributions. Chapter 3 provides an overview of type systems while providing several examples. Chapter 4 introduces the Espresso language. A formal specification and corresponding type system for the Espresso language is briefly presented here − only the essential core of the language using an abstract syntax of Espresso. In Chapter 5, we provide an overview of *Jiapi*, our type checker generator tool for the Java-like classroom language Espresso. We have included some common definitions and notational patterns for the reader's convenience. Chapter 5 also describes the implementation of *Jiapi*; in particular, we illustrate how to apply it in different parts of the type checker specification. Chapter 6 introduces case studies involving two imperative and object oriented languages. Chapter 7 compare existing relevant work. Chapter 8 concludes with the remaining activities for the finalization of the thesis.

Chapter 2

Problem Statement

2.1 Introduction

In computing, the problem of correctness is *always* ubiquitous: A program is developed with a certain specification in mind, and it is run under the assumption that it meets that specification. This assumption, as we all know, is frequently unjust: in the vast majority of cases, the program fails to operate as it should. But what should be done about this problem? Only a formal proof of correctness can guarantee that a program meets its specifications [102, 103, 104]; testing cannot guarantee the absence of mistakes [105], but it can *only* demonstrate that a program is not correct. So far, if we take a simplistic approach to this strategy (e.g., testing programs), in which we write the program and then provide a proof that it complies with a specification thereafter, there still is a risk that the program we develop may *not* work properly. We should instead aim at developing the program in such a way that it must perform as it ought.

In 1972, Edsger W. Dijkstra addressed the issues posed by the lack of program correctness proofs and devoted a few lines to the problem:

> "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would

only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand." [106]

Essentially, Dijkstra is proposing the approach taken by advanced type theories, of allowing programmers to describe important semantic invariants of a program in a type system [107, 108, 109], in which we can describe a family of constructs spanning a set of parameters, each of which is unique to the constructs being specified. The type system is therefore built in such a way that a program is only adequately typed if the required invariants are satisfied. In light of this, type theorists believe that given a powerful enough type system, most − if not all − characteristics of a language that one wishes to preserve can be proved for certain programs. Even so, even if the core mechanics of a program are well understood, implementing a new type system can be a challenge.

The problem with the average programmer is that type systems are often designed using axioms and inference rules [110], possibly with various side conditions, to prove the absence of certain kinds of program behaviors; for example, writing logic formulas as *pre*– and *post*–conditions, and verifying them mechanically. This leaves average programmers with no choice but to perform the proof equivalent of bookkeeping, as many of them are unfamiliar with proof-theoretic definitions. That is, the most basic version of this technique entails feeding numerous inputs to the tested program and checking the result for correctness. Other more advanced versions of this technique attempt to select inputs in such a way that all, or at least the majority, of the available execution paths are examined. Of course, this is a shaky notion of program correctness, but it is quite useful in identifying errors in practice.

Strictly speaking, type checking is a lightweight technique for demonstrating that a program satisfies specific requirements. Unlike theorem-proving techniques based upon axiomatic semantics, type checking rarely determines whether a program will yield the correct result. Instead, it is a means to test if a program is well-formed, using the premise that a well-formed program meets desirable properties. This implies demonstrating that a type-correct program might never reach a non-final configuration in its operational semantics where its behavior is unknown [25, 24][8]. Our solution to this challenge is to give programmers the tool

[8] A well-typed program cannot become stuck in the middle of a computation; in other words, well-typed programs cannot fail [86].

to *test* or *reduce* a proof to a limited number of easily-reviewable predicates. For example, we might find helpful to try to prove the theoretically equivalent interpretation of a desired property of a program, in which we simplify things for the type checker, such as by breaking up various cases or adding predicates to break complex proofs into smaller steps.

It should be emphasized that, for a large range of statically typed languages, it is impossible to develop a type checker that accurately distinguishes between ill-behaved and well-behaved programs due to the theoretical limitations of type systems [35, 36, 39, 87, 111, 112]. For example, statically checks can be overly burdensome if they require a lot of bookkeeping without sufficient proofs. This raises the following question: How effective are type systems in the real world? It turns out that they differ markedly when comparing their notion of type and genericity, from very simple (even though efficient to compute, they are also quite restricted) to quite sophisticated (even though difficult to compute, they can be incredibly expensive). Needless to say, a substantial part of current programming language research focuses on improving the power and expressiveness of type systems while keeping them practical.

2.2 Thesis Statement

As stated above, the challenge in constructing a type system is to find the right balance between soundness, completeness, and decidability. There will always be correct programs in that they will never crash at run-time due to data misuse, but we will never be able to convince the compiler that these programs are correct. For example, they could rely on complex invariants or subtle data and control-flow dependencies that the type system is not equipped to detect and exploit (e.g., [35, 36]). We could try to improve the type system to do this, but that would require the programmer to provide enough evidence at all times to satisfy the new type system, and that evidence (in the form of type annotations) may be incredibly tedious [55, 113, 114, 84]. Worse still, some type safety properties may not even be decidable.

Unfortunately, even with today's tool support and approaches, creating mechanized type soundness proofs is a demanding task for verification professionals. The provided assistance

frequently entails manually spelling out a huge number of *trivial* stages inside such proofs, requiring a certain degree of skills and competence. The thesis's ultimate goal is to increase the degree of automation for mechanizing type checking in a simple yet practical way. The idea is that a value is regarded as a proof of the statement corresponding to its type, while the type is interpreted as a proposition, resulting in the Curry-Howard correspondence [115][9]. As we shall see later, this concept will be used to derive the majority of common logical connectives. To this end, we present a framework for implementing type checkers that adheres to common design patterns. Although we focus on domain-specific languages (DSLs), we use Espresso (a large subset of Java) as an example.

The type system we will develop here involves not only primitives but also constructed types. Given that Espresso is object-oriented, we will need to consider classes and other sorts of reference types that are related to the basic concept of a class (such as interfaces and enumerations, and in some cases, procedures and functions[10]) but differ in some way and are typically treated differently by other languages. Hence, understanding how the basic structure of Espresso's type system works will lay the groundwork for understanding how *Jiapi* can be used for developing more advance systems.

To reduce the possibility of errors, we will structure all of our validations as follows:

- **Formal validation:** Takes the form of a sound semantic model − or, to put it another way, a logic model − based on a set of judgments (i.e., predicates about a language's properties) and a set of rules (i.e., implications between judgments). It is worth noting that this description consists only of *rules*, with no *axioms*, because rules are what allows us to conclude one judgment from a collection of others. Now, formalizing the syntax and semantics of predicates, and thus formalizing the set of rules, which are just set notation and first-order logic, require four parts:

  (*i*) A general statement of what we wish to prove.

[9] The Curry-Howard correspondence is a theory that answers questions such as *Can a program compute a value of some type given values of some other types?*, *Is it possible to deduce a function's code from its type signature?*, etc.

[10] Procedures and functions can be treated as values in some languages (e.g., Algol [116], Simula [117], Scala [118], Python [119], Perl [120], JavaScript [121]), allowing them to be assigned to variables or given to other procedures and functions.

(*ii*) A description of the set on which the prove will be performed.

(*iii*) One or more particular cases that represent the most basic case.

(*iv*) The steps we assume to exists. In general, we will assume that the predicates hold (i.e., these are statements that we can prove, disprove, assume, negate, and so on), but sometimes we will need stronger assumptions. For example, the predicate $numeric_?(\tau)$ could be defined as a disjunction of other predicates: $numeric_?(\tau) = byte_?(\tau) \vee short_?(\tau) \vee \cdots \vee double_?(\tau)$[11] (see [5, Section 7.2.4]). Alternatively, for atomic types, we may define this predicate as $numeric_?(X) = X.type \in \{int, long, float, double, \ldots\}$, where $X.type$ refers to the named attribute *type* of node X (see Section 5.3 on page 112).

Each rule, in addition, will have a set of premises and a conclusion, and can act on any Java object that represents a type[12] – rules typically act on the objects representing parse-tree nodes. For the set of rules, we prefer that users utilize accurate mathematical notation (see Section 5.2 on page 107). If a user does not know how to write in mathematical notation, we will accept semi-formal statements in plain English as long as they are correct, unambiguous, and complete.

- **Practical validation:** Comes in the form of several worked examples exhibiting different aspects of the compiler. These will include both program that compile and ones that should not. In general, a successful set of test cases checks if the implementation follows the design, whether the design contains loopholes or ambiguities that allow for erroneous usage, and whether the implementation's behavior can be predicted in all scenarios. Upon running, this set of test cases *should compile* and *should pass*. It can also be noted that writing tests after writing the specification will provide us with insight into what our implementation should accomplish. Moreover, it will assist us

[11] Naturally, a compiler or interpreted for a typed language (or intermediate representation [122] like the Java bytecode) may require that a well-typed input satisfies a predicate. For example, in the implementation of a type checker, we can introduce one predicate function for each atomic type, one for each type constructor, and some to detect whether a type is integral or numeric.

[12] Note that subclassing and subtyping are synonymous in most object-oriented language type systems. However, in our type system, types are defined by classes, with each class defining a type or a family of types.

in sorting out the types that we have in the language, and often understanding that structure is a tremendous help in understanding the problem.

We believe that by suggesting a division of aspects, objectives, and activities, this method will make language maintenance easier. Such a division will also allow DSL advancements to be integrated, as feature requests and requirements change often to meet the regular expanding needs.

2.3 Why Static Typing

The world itself is **typed**. We cannot add or subtract lengths in weights, and although we can add or subtract lengths in inches (decimal or fractions), feet, centimeters, and millimeters, we should convert at least one of the two. Failure to do this could bring *any* Mars journey to a halt in a literal sense (e.g., [123, 124, 125]). On the other hand, in a language with a static type checker, adding two lengths expressed in different units would have resulted in an error or an automatic type cast. Consequently, seeing the importance of types in providing a foundation for intrinsic code documentation (such as useful variable names, highlighting keywords, or anything affecting the actual code), it is good practice to try to choose languages that support static analysis and type checking.

Indeed, any type system must be able to address at least some of the issues listed in Section 1.2 on page 5. In a statically typed language, it is common practice to use a type checker to filter out possibly ill-behaved programs before they are executed [3, 5] – specifically after the program is read in and parsed but before creating the executable file. For example, instead of lingering undiscovered until the code is deployed, possible errors can be detected during the compilation process. This is because programs must abide by the restrictions imposed by their types, or they risk being rejected. In plain words what this means is types are about expressing more of our intent to the machine so it can do more work for us. It is for this reason that statically typed languages can guarantee the absence of entire classes of errors in programs [126, 127].

What follows is a summary of the benefits from adopting static typing in as few words as possible. Note, however, that none of these points are new or unique, but we wanted to

consolidate them in one place to explain why we decided to create a tool for statically typed languages. (Make reference to [128] for how static typing improves the maintainability of software systems.)

- Untyped programs can often be unreadable, unreliable, and inefficient [129, 130]. For example, we could prove that executing a program written in an untyped language results in a type error, but we would have to do it for every program written in that language, whereas in a typed language, it is guaranteed in every case.

- A substantial number of errors are discovered earlier in the development process, closer to the point of introduction. For example, consider a function f that accepts one parameter of type *string* and returns a value of type *int* (the type of f is $string \rightarrow int$), but we accidentally pass an *integer* instead elsewhere in our program. With static typing, the type checker would catch this issue and alert the programmer to the incorrect value being given.

- Types guide development and can even write code for us, reducing the amount of information that the programmer has to specify. For example, the types of variables may be inferred from context by the compiler as type inference provides the programmer with an option: The freedom to choose between manifest and inferred types. Although a number of compile-time analyses requires reasoning about unknown types, in object-oriented languages like Java, C++, and C#, the initializer (e.g., constructor, literal, method return value, etc.) *always* provides all of the information for an inferred type.

- Refactoring can be done with more confidence because a substantial number of errors introduced during refactoring will end up as type errors. For example, consider the same function f that returns a single value of type *integer*. If we wish to change it to return a list of numbers instead, the compiler will catch most − if not all − the places that need updating after modifying the type signature of this function.

- If we used static typing, we are prevented by the rules of a language's type system from forming expressions that may result in type errors when the program is executed.

Therefore, types can make developing programs easier by automatically tracking information that a programmer would otherwise have to keep track of mentally. For example, Dropbox developed a tool for performing type checking on interpreted languages [131]. Their major argument was that coding at scale means ensuring that developers can quickly understand the present codebase and become productive by using the power of types and reassurance of type checking (e.g., [132, Section 3]).

- Type annotations serve as code documentation that is guaranteed to be valid, unlike programmer comments, which are frequently missing, erroneous, or out of date. This is especially true for many tasks that programmers often take for granted in their IDEs and toolkits (e.g., IntelliJ [133], Eclipse [134], and Netbeans [135]). For example, static code analysis, refactoring, code simplification, inlining variables, point out locations with code duplication, to name a few, can be performed by IDEs and other tools by means of types, type information, and type checking. Thanks to these types of knowledge, a tool in our development environment (e.g., a *daemon*) may analyze our code and immediately flag areas where we can enhance it.

- When types are used as annotations (not to be confused with the preceding bullet point), they can be read by both programmers and the compiler, which may be able to provide a "safety guarantee". For the compiler, the safety guarantee will usually specify that operations are only performed on arguments of the correct types, thus preventing or reducing the frequency of run-time errors. For the programmer, types can aid in the definition of more complex analyses, the reasoning of an analysis' soundness, and the efficiency of static analysis [136]. For example, using static analysis for finding dynamic errors [137], or finding and handling bugs in system code, such as Linux file systems and drivers [138].

- Finally, type information eliminates the need for run-time type checks and allows the compiler to optimize the entire code for better performance. For example, execution can be 10 to 100 times faster than dynamic languages [24], becoming a clear advantage for programs where efficiency is crucial. (See also [139, Section 3] for static and dynamic optimization techniques for object-oriented languages.)

It is worthy of note that the strictness with which *static semantics*[13] are enforced varies substantially between languages. In actuality, none of the above is enforced by all programming languages. All of these factors compel us to formally specify the semantics of the language we are implementing, even if we already have perfectly adequate implementations. For example, several papers have been written about ProcessJ (including [21, Section 5.1], [140, Section 1.1], [141, Chapter 5], [142, Chapter 3], and [143, Section 4.4]) with regard to combining the familiar syntax of Java/C++ with the process semantics of occam/occam-π. In general, the more static checking there is in a compiler, the less manual debugging is required (e.g., [144]).

2.4 Motivation

As we all know, erroneous program behavior is all too common; nevertheless, without observing wrong behavior, we cannot generally state if a program is well-typed or whether we simply have not tested it with an input that would cause an error. Furthermore, type soundness is no longer trivial after a language and its type system are enhanced with additional features. Even though each feature may be *sound* in isolation, the combination of two − or more − features can be *unsound*. This is troublesome since researchers often prefer to look at each element in isolation and do not always anticipate issues with the combination. For this reason, as software becomes increasingly complex, employing more powerful methods for verifying that a program is well-typed (those that can provide the highest level of certainty) becomes exponentially more difficult for the average compiler writer and programmer.

Part of the reason is that the structure and semantics of types have been formalized and reasoned about using type systems, which is done to ensure that type declarations and expressions are *sound* [114]. Although there exists different formalism to specify semantics, operational semantics notation is often used as a basis for semantic specifications. The semantics are expressed in the form of axioms and rules, consisting of a collection of premises and a conclusion that form the basis of a proof. These rules are closely related to a language's

[13] These are semantic rules that can be checked prior to the execution of a program [24, 25]. For example, if we verify the type of expressions in advance of execution to forecast some features of the program's dynamic behavior, then this type checking is part of the language's static semantics.

abstract syntax, as we have briefly demonstrated in Chapter 1, and therefore can be explicitly divided into smaller rules in order to ensure type safety for declarations and expressions. Driven by this, we decided to create something more practical.

The following bullet points influenced our decision in the design and development of *Jiapi*:

- To ensure that a type system is well defined, various formalism are utilized; however, type checkers in commercial language processors rarely use tools to transform these formalism into code. For example, the majority of existing compiler and interpreter development tools are focused on the creation of scanners and parsers.

- Type systems are defined and written precisely. The verification of formal properties (such as type safety) in detail provides assurance that their definitions are correct [145]. Russell's initial ramified theory of types [146], Ramsey's simple theory of types [147], Church's typed lambda-calculus [148], Löf's constructive type theory [66], and Baerardi's pure type systems [149] are all notable milestones in type systems formalization. These verification models formalize data − and data abstraction − and give techniques for describing structural constraints; however, their perceptions are vastly different and somewhat difficult to apply even for individuals who are experts in the area of mechanized verification.

- Proving that a program is well-typed is also limited to demonstrating that a program adheres to a set of specifications and requirements, namely, the type system of the language. But what if the specification − of the type system − has a flaw? This is at the center of the critique in *Social Processes and Proofs of Theorems and Programs* by de Millo, Lipton, and Perlis [150], arguing that formal program verifications will not play the same crucial role in the advancement of computer science and software engineering as proofs do in mathematics, regardless of how they are attained.

- Often, the proofs are frequently extensive and tedious proof-by-cases, with less engaging intellectual content than most mathematical proofs. For example, such proofs employ formal verification schemes to ensure that the system's fundamental principles are correct before they are accepted [151]. In fact, most languages do not have a fully

formal specification − or are not fully formalized mathematically − outside of research contexts.

- Finally, formal semantics is needed for whichever language a program is specified in − *all languages have semantics independent of their implementations* (c.f., [152, 153, 154]). With today's tools, describing the formal semantics of even a simple language is a big undertaking, both conceptually, in the form of mathematical frameworks, and with software, such as theorem provers. (Isabelle [155] and Coq [156] are two examples of powerful proof assistants.) For example, there may be possible input values that require special handling or strange compiler behavior that is simply difficult to express and model, requiring a certain level of semantics maturity that many do *not* posses.

For these reasons, we believe that we should establish more realistic targets given that we can solve nearly no general problem in the realm of program correctness (in this instance, well-typed programs); for instance, there is no algorithm for determining if a given input will cause an arbitrary program to terminate [157, *The halting problem*, Section 5.1]. Fortunately, the lack of generic solutions does not rule out the possibility of proving well-typed programs in specific circumstances or in a context that is confined in some way. We *strongly* believe that we can focus on actually solving the problem at hand once we have established the importance of well-typed programs as an engineering and theoretical problem.

2.5 Contributions

This thesis makes a significant contribution by providing a new perspective on the problems of **type checking** and **type inference**. The main contributions of this thesis are as follows:

1. Demonstrate that if a type system is described in a formal way as part of a language development, the tool ships the potentially implementation of a type checker.

2. We created a simple formalization for capturing type system specifications using *set notation* and *first-order logic*. This formalization is quite expressive and incorporates modern programming features such as recursive types, inheritance, polymorphism, and

exceptions. It also provides a clear separation between atomic types and constructed types, as well as subtyping and subclassing if applicable to any reasonable DSL. Thus, we can arrange and constrain a language's type system specification in a straightforward and unified manner without having to modify our approach for each new programming language. Moreover, we created a meta-language for describing data types with a minimum set of required characteristics. Under this circumstance, we can use simple elements (i.e., characters) and constructs (i.e., a combination of characters and symbols) to describe the definition of records/structs, enumerations, classes, functions, and other patterns (Table 8, in Section 5.2 on page 107, summarize punctuation and keywords in *Jiapi*).

3. A tool that takes a type system definition as input and generates a specific implementation of a type checker for any reasonable DSL. The tool is accessible to the compiler in the form of a tree API that provides an *object based* representation of a type checker. This type checker works as a function that converts the type of an expression from an AST representation. For example, a type T is represented by a tree of objects in this object-based model, with each object representing several *type system nodes* (such as atomic and constructed types) and each object having pointers to the objects that represent its constituents. Thus, if y is a *double*, the AST representing the expressions $y - 2.5$ will be mapped into the data structure representing the type *double*. It is worth noting that, because most languages support many binary and unary operators, hard-wiring their type checking into the type checker with code is tedious. Hence, in order to make the type checker easier to update and maintain, we use a special symbol table to hold all operators (as well as system functions) along with their signatures. Finally, the tool creates the necessary Java files, which must be imported into the source code for compilation.

4. Finally, we can demonstrate that type system specifications for languages that conform to our meta type systems are guaranteed to be valid using empirical data. This instills a high level of trust in our typing systems.

Chapter 3

Type Systems

3.1 Introduction

A compiler must do more than simply recognize input and to respond to it in some way such as being able to properly identify different components and differentiate phrases in a language. It must be able to determine useful information for later phases and find errors that would render a program invalid. In other words, if the program is not semantically valid, it will not be possible to generate code such that the meaning of the target program is the same as the meaning of the source program (albeit, as we will see later in the chapter, this is **not** true for any useful object-oriented language). Following parsing (or *syntax analysis*), the next phase of a typical compiler is *semantic analysis*. Fig. 5 illustrates the elements and basic data flow of a compiler's front end.

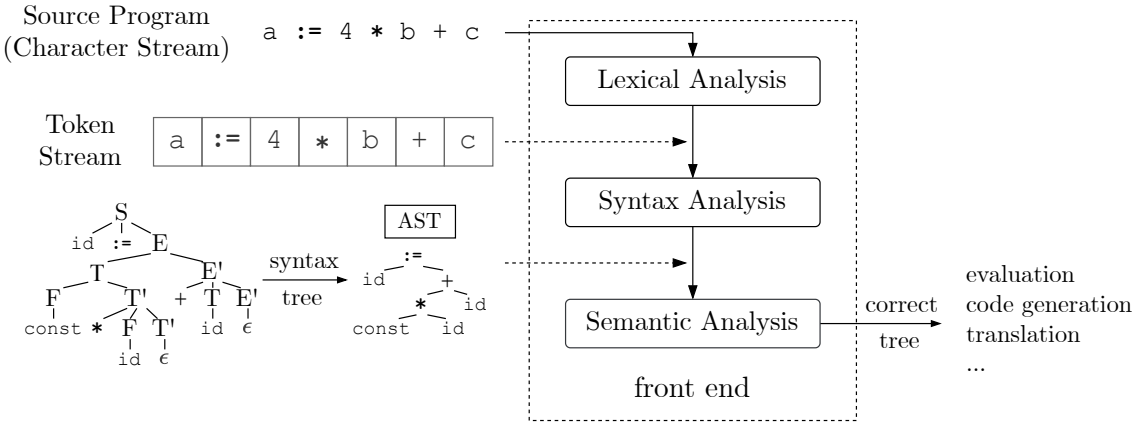


Figure 5. Basic data flow of a front end.

A variety of checks are performed during semantic analysis to ensure that the source program can be translated into the intermediate or target code [5, 158]. Once the syntactic structure of the program is known, semantic analysis computes additional information related to the program's meaning. This is done to allow for semantic checks, such as checking for type errors, associating variables and functions references with their definitions, requiring variables to be initialized before use, among other things, in order to ensure that a program is sound enough to proceed to code generation. Therefore, to be *semantically valid*, all declarations and statements in a program must be appropriately defined, expressions and variables must be clear and consistent with how control structures and data types are expected to be used, and access control must be respected.

Typically, semantic analysis involves keeping track of variables, functions, declarations, and type checking. For example, a language may require identifiers to be specified before they can be used. The type information assigned to these identifiers is then recorded as the compiler encounters new declarations. This ensures that the type of an identifier is respected in terms of the actions executed as the compiler continues to inspect the rest of the program. The language may also require that identifiers be unique, making it impossible for two global declarations to have the same name. Arithmetic operands may be required to be numeric, perhaps even of the same type where no automatic data conversion is allowed (e.g., no *int*-to-*double*). Finally, it may also be necessary for the right-hand side of an assignment statement to match the type of the left-hand side, and the left-hand side to be a correctly defined and assignable identifier. These are some examples of the things that the semantic analysis step looks at.

In order to perform these checks, the **type system** of a language must describe which operations are valid for a type. As a result, the aim of **type checking** is to ensure that operations are *only* performed on variables − and expressions − of the correct types and number. Normally, in object-oriented languages, type-checking is accomplished by a *visitor*[14] calling a set of methods at the proper times, and as the syntax requires. All operators are type-checked to ensure that they are compatible with their operands. We refer to operations

[14] A *visitor* provides an iterator over a tree-like data structure. Each node in a parse tree is, in general, an object instance of a subclass that represents a specific node type. Each such subclass − of the general *visitor* class − reimplements a particular method for each node that requires any action such as a visit.

such as addition and multiplication as well as explicit assignment using the assignment operator (=), and implicit assignment of arguments to formal parameters in function calls.

We begin this chapter with some basic definitions that will help in setting the stage for performing semantic analysis in Section 4.4 on page 62. We explain what a type system is, how it is specified (in accordance with type safety, type information, type inference, and type annotation), how it defines types for language constructs, how it can be implemented by a type checker, and introduce some of the basic issues with types and type checking.

3.2 The Big Picture

When writing code using **statically typed languages**, we always know something about the types of the values we are working with; otherwise, we would not be able to write meaningful programs. When looking at a data structure or interface, it is also helpful to be able to discern which data is unlikely to change, which data is likely to change, and who might change that data. Basically, the data types that the data source provides in a program (i.e., the primary location from where data comes), the semantic category that matches these data types, and how they map to the data types that a language supports are all determined and specified by the integrated type system.

In most mainstream statically typed languages, a programmer must first define the *name* and *type* of any data object, and, in most cases, the programmer also determines its lifetime. A declaration, for example, is a statement in a program that provides the compiler with this information. The most simple declaration consists of only a name and type, although in some languages, *modifiers* that control visibility and lifetime may be included (e.g., Java and C++ classes). The following C++ statements are examples of declarations, visibility, and lifetime.

```
void print(const Array& a); // function prototype

int cnt = 0; // global variable avaiable

class Array {
```

```
  // variable only accessible within the class
  std::vector<int> list;
  // this access modifier makes everything from here on
  // available and accessible in main
  public:
  Array(int size) : list(size) {}
  // a const reference qualifier in a member function
  int operator[] (int idx) const {
    return list[idx];
  }
};

int main() {
  // local variable available only in main
  Array a(10);
  ...
}
```

Additionally, functions also have types – the type of value they return and arguments they take, which the compiler uses to ensure that a program calls a function correctly. In C++, for example, function *prototypes* are comparable to variable declarations in that they both serve the same purpose. In a function call, the prototype is used by the compiler to check the number and type of arguments. The location and qualifiers, however, establish the visibility and accessibility of the function (e.g., is the function local to a file or global?, is it defined in a module, a package, or a namespace?, is it nested in another function?, is it defined in a class?, etc.).

While a type qualifier is used to improve the declaration of variables, functions, and parameters in C++[15], a *qualifier* in a declarative language like Analytica [159] ensures that a formal parameter receives the kind of value that a function expects. Consider the function `Sum` with parameters specified as

[15] When using a type qualifier, we can specify whether:

- The value of an object can be changed.
- An object's value must always be read from memory rather than a register.
- A modifiable memory address can be accessed by several pointers.

Note that in this context, the term *object* refers to data, not an instance of a class.

```
Function Sum Parameters: (a: Number; b: Number)
```

and which may be called in an expression like this

```
Sum(4, 6)
```

The parameters `a` and `b` in the `Parameters` attribute (termed formal parameters) require that the actual parameters 4 and 6 evaluate to numbers as specified by the qualifiers `Number`; otherwise, it is considered an error.

In reference to other declarations and uses, *type declarations* such as C's `typedef`, C++'s template expansion, and Java's generic type declarations behave in a similar way. For example, a `typedef` in C creates an alias or a new name for an existing type or user defined type. We can write a `typedef` declaration anywhere other declarations are allowed, but the scope of the declaration depends on the location of the `typedef` statement. Consider the following C snippet of code:

```
int main() {
  typedef unsigned char uchar; // only avaiable in main
  uchar c = 'b'; // interpreted as unsigned char c = 'b'
  ...
}

uchar ch = 'a'; // error due to unknown type
```

Since the `typedef` definition is declared inside the `main` function, the scope is local, and the alias can only be used by the function that contains the `typedef` statement. Otherwise, an error occurs during compilation as the new name (i.e., the alias) is unknown.

In parameterized classes or generics [160], a *generic type* is a type having formal type parameters. A paremeterized type is a generic type that is instantiated with an actual type arguments (i.e., it means binding its parameters − its type variables − to actual types). In order to use a generic type in Java or C++, we must provide one type argument for each type parameter that the generic type declares. Consider the following C++ snippet of code:

```
// array.h
template <class Type>
class Array {
  Type *list;
  ... // same as before
};

// main.c
int main() {
  // instantiation of the generic class Array
  Array<int> arr(10);
  ...
}
```

Before instantiation, a template declaration and definition is examined for syntactical accuracy. The compiler then instantiates the generic type in order to verify the concrete parameter types and non-type parameter values before applying the constraint checking templates to these types. As an example, a template definition may occasionally use names that are not defined by the template arguments or within the template itself [161]. If this is the case, the name is derived from the scope enclosing the template, which could be the context at the time of definition or the context at the time of instantiation. Furthermore, since template instantiation relies on *type-name equivalence*[16] to determine which templates need to be instantiated or reinstantiated, local types can cause problems when used as template arguments. Consider the following C++ code:

```
// file1.h
#include "array.h"
struct A { int a; };
Array<A> x;
...

// file2.h
#include "array.h"
```

[16] C/C++ uses structural equivalence for everything, except unions and structures, for which it uses loose name equivalence.

```
#include "file1.h"
struct A { float a; };
Array<A> y = x;
```

the `A` type defined in `file1.h` is not the same as the `A` type defined in `file2.h`. This indicates that the assignment expression `Array<A> y = x` is considered an error (see also [73, 162, 163] for a detailed discussion of class and function templates, along with examples).

Ultimately, we want a language that is safe, with a few exceptions, and also typed. We would also like the language to be **sound** if it is typed, and to have a type systems that is **decidable** so we can be sure that the types are not "lying" (see Definition 1). For example, when we write `int y`, we hint that `y` will always contain an integer value, and that other sections of the program that rely on `y` can trust that this statement will be enforced. Of course, all this form of knowledge can be encoded, manipulated, communicated, and checked using a type system. Thus, when creating a type system for a language, the language designer must also strike a balance between execution, efficiency, expressiveness, safety, simplicity, and other factors like *typedness* and *soundness*.

3.3 The Role of a Type System

Probably, one of the main significant points of difference between programming languages is their **type systems**. Even languages that appear to be similar in "appearance" have vastly distinct type systems. As an example, while C and Java do not have the same set of types, they do have similar syntax and control structures; however, the rules that determine whether or not a program is legal with respect to types are significantly different because each has a distinct type system. Consider the following snippet of code:

```
int p = (int) "Should this be legal?";
```

Java's type rules does not allow the above statement because *String* and *int* are not in the same type hierarchy (that allows for data type conversion), but C does allow it. Then

why do different languages use different type systems? Well, this reflects the fact that there is no such thing as a *one-size-fits-all* type system. Each type system has its own set of advantages and disadvantages. As a result, different languages employ distinct type systems according to their differing objectives. For example, since C was originally derived from the typeless language BCPL[17] [71], it seems reasonable to cast from `(void *)` to any type without having to explicitly cast so. This is particularly important when allocating memory dynamically in a *malloc*, *realloc*, or *calloc* call, or when accessing and manipulating any kind of data. Thus, due to the implicit narrowing conversion in C, it is possible to convert memory addresses to integers since a pointer to *void* may be converted to or from a pointer to any other − incompatible − type [165].

It should be clear that a strong understanding of type systems is essential for understanding how to get the most out of programming languages. A type system is defined as follows:

Definition 2 (Type System): A *type system* is a collection of types, along with a set of rules for associating types with expressions, variables, and other entities that can have a type. [5] □

In terms of programming languages, type systems refer to a set of rules that, when applied to language terms, generate types for those terms [4, 166, 114]. Essentially, a type system is a method of classifying entities in a program − such as expressions, variables, functions, and other entities as described in Definition 2 − according to the types of values they represent in order to avoid undesired program states; namely, the type of an entity is the *classification* attributed to it. For example, consider the following Java code:

```
System.out.println(7 + 3.14);
```

Although this does not need to be explicitly stated, literals `7` and `3.14` represent values of type `int` and `double`, respectively. When applying the binary operator `+` to two values of type `int` and `double`, the result yields a value of type `double` because of the data type

[17] BCPL [164] was a typeless procedural, imperative, and structured programming language intended for writing compilers for other languages.

conversion between Java's numeric data types (Figure 6 illustrates how Java automatically coerces data in the direction of the arrows[18]). Because one operand of a binary operator is `double` and the other is not, the non-`double` operand is converted to `double` before the operation is performed. Therefore, Java's static type system assigns the expression `7 + 3.14` the type `double`.

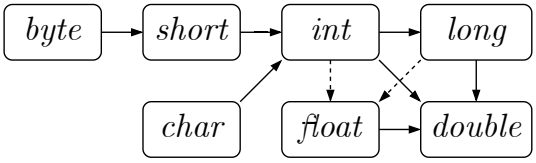


Figure 6. Java's numeric data type conversion.

Furthermore, any statement that tries to coerce data against the direction of the arrows will be flagged as an error by the compiler. For example, the Java compiler issues a "possible lossy conversion from long to int" as an error message for the following declaration:

```
int num = 100L;
```

In the past, data could not be specified in terms of other data types because it was common for programming languages to have a fixed set of types [167]. Very often, these languages imposed a restriction on the sets used to create types, requiring that any operation associated with a type has to act consistently across all values of the type. For example, the set $\{-2147483648, \ldots, -2, -1, 0, 1, 2, \ldots, 2147483647\}$ denotes a type since addition, subtraction, division, and other operations could be applied equally across all values. The set $\{\texttt{false}, \texttt{true}\}$ also denotes a type since conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$) could be applied equally across all values. However, the set $\{3.14, \texttt{false}, \texttt{"seven"}\}$ does not denote a type since there are no operations that could be applied equally to the values of this set.

In contrast to old languages, specialized and modern languages generally contain an **extensible type system** [168] that allows programmers to specify new types, new operations,

[18] Coercions that follow a 'dotted arrow' are permitted, but they may cause loss of precision, which means the converted value may not be equal to the original value. Despite this, many languages allow type coercion as a convenience to the programmer.

new notations, and, on some occasions, new verification schemes. For example, defining a `class` in Java constitutes creating a type like this:

```
class A { ... } // where A represents a new type
```

Moreover, it is possible to be completely non-descriptive when creating a "set" of different values since *Object* sits at the top of every class hierarchy in Java. For example, in

```
Object[] list = {1, 1.2, "3.14"};
```

`{1, 1.2, "3.14"}` denotes a type because there may be operations (e.g., the + operator can also be used to concatenate strings while defined over numbers) that could be applied equally across 1, 1.2, and "3.14". This is because 1 and 1.2 are **autoboxed**[19] to *Integer* and *Double*, respectively, which extend the superclass *Number*, which extends *Object*, and *String* also extends *Object*. However, treating this array as an array of integers is unsafe because someone reading from the array of integers would expect to obtain integers but could instead get arbitrary objects.

Figure 7 demonstrates an illustration of the associated class hierarchy for each element in the list of objects. Note that the class hierarchy is a *tree*.

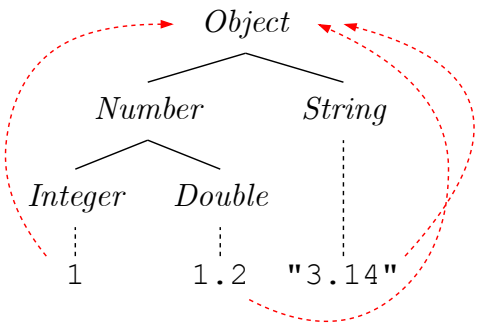


Figure 7. The class hierarchy for elements in a list of objects.

Looking at the class hierarchy in Figure 7, if we were to omit the type `list` has, and we wrote `...[]` instead, what would the element's default type be? If it is *Object*, do we

[19] *Autoboxing* in Java is the process of assigning a primitive value directly to a wrapper object without calling the wrapper object's constructor [169].

always need to downcast to integers or doubles? If it is *Number*, how do we guarantee that we will only add two elements of the same or equivalent types? As we will see later, the situation for type checking is not as favorable in object-oriented languages. Although great progress in building static type systems for object-oriented languages has been made (mainly as a result of research on type systems [170]), some still remain unsafe or inflexible [171], requiring the programmer to "guide" the system to check for types that it cannot detect, forcing the usage of **type casts** to compensate for the system's deficiencies. These type casts can either be unchecked (like they were before the introduction of `dynamic_cast`[20] in C++) or checked at run-time (like in Java). Even when contemporary languages like Java [172], C++ [173], and C# [174] have more strict type systems than desired, others like Eiffel [175], Groovy [176], and Kotlin [177] require dynamic or link-time checks to guarantee *type safety* and ensure the integrity of the computation.

3.4 Type Safety

Type safety is an important property in programming languages as it can prevent programs from causing errors. A type safe language − one whose programs are known to contain no **type errors**[21], at the very least, ensures that its programs have a well defined meaning. This guarantee is necessary for reasoning about what programs might do, which is especially significant when security is an interest. As such, a language is said to be **strongly typed** if all type errors can be detected at compile-time before the statement in which they can occur is executed [5, 158]; that is, no type-related errors can occur at run-time. However, if *any* checking is deferred to run-time, the language is said to be **weakly typed**. Some languages, like C, provide explicit pointers on which arithmetic can be done. These languages are referred to as weakly typed since expressions with pointer arithmetic cannot be type checked at compile-time (e.g., *null* dereferences, dangling pointers, leaking memory, and unintentional aliases are all potential hazards).

It is helpful to remember Cardelli's terminology in order to be able to explain what con-

[20] The `dynamic_cast` operator allows donwcasting of polymorphic types; it casts data from one pointer or reference type to another, performing a run-time check to ensure the validity of the cast.

[21] A type error occurs when a program tries to execute an operation on a value for which the operation is undefined.

stitutes program errors [4, 62]. Different types of execution errors[22] (or run-time errors) can occur when a program is executed. These errors can be divided into two groups: *untrapped* and *trapped* errors. Untrapped errors are errors that can go unnoticed for a while and then cause arbitrary behavior. For example, in C++, reading data beyond the end of an array or jumping to the wrong address may not create an immediate error, but it may cause the program to crash later during execution. On the contrary, trapped errors are errors that cause a computation to stop immediately. For example, in C++, dereferencing of a *null* reference, accessing an illegal address, dividing by zero, or trying to read input from a file that does not exist will cause the operation to stop. However, these errors can be handled by the run-time system or by a language construct, such as exception handling (i.e., a *try-catch* block). Examples of errors that fall somewhere in between these two extremes are *subtype-violation* errors (SV for short). These are the kinds of errors that occur when values do not match a declared type at a write operation [178]. Take, for example, the following snippet of code from [178], where in *checked mode* (also known as the *debug mode*), the assignment in ❶ will cause the program to stop, but the argument in ❷ will cause the program to stop while in *production mode* (also known as the *release mode*):

```
String x = 5; ❶
print(x.length); ❷
```

Undoubtedly, understanding the difference between these two types of execution errors (that is, *untrapped* and *trapped*) can aid in reducing the number of typing errors in a program and making it type safe. A simple categorization of program errors is shown in Figure 8, where program errors $\in U$, trapped errors $\in A$, untrapped errors $\in B$, errors caught by safe languages $\in E$, and execution errors $\in A \cup B$.

[22] Execution errors frequently cause a program to crash, whereas non-execution errors may cause a program to provide incorrect results due to wrong logic.

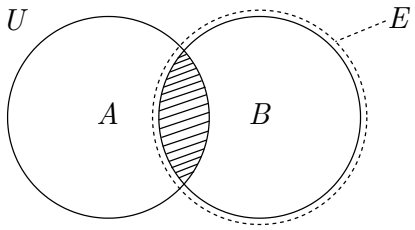


Figure 8. Venn diagram of programming errors.

Execution errors are a subset of all program errors (or bugs) in this diagram. Since untrapped errors are obviously the most pernicious type of execution error, a language's type system should aim to catch as many untrapped errors as possible (e.g., **safe** languages are those that eliminate *all* untrapped erros). To minimize execution errors, statically typed languages add a *type checker* to ensure program safety by preventing an unsafe program from running. These checks can be done as *compile-time checks* (also known as *static checking*), in which the compiler detects possible errors and reports them to the user before creating an executable file. However, some errors, such as array index out-of-bounds errors or down-casting (i.e, from a supertype down to a subtype), are difficult to address statically, so many languages perform an additional *run-time check* (also known as *dynamic checking*) as part of the program execution.

Whether at compile-time or at run-time, the main objective of a type system is to reduce the number of execution errors that can occur in a program. What matters is whether or not a given operation excludes any values. If it does, then we can consider if the language prevents the operation from being used with any values that are prohibited. If the language does *not* allow it, we label it as safe; but if it does, we label it as unsafe. Indeed, a language can be safe for some operations while being unsafe for others, and so we generally refer to a language as "safe" if *all* of its operations are safe (hence, we say that the language is type safe if an operation does not lead to undefined behavior).

3.5 Type Information

In addition to catching errors, *type information* can also be used for type checking. Most current compilers find and generate a variety of information during compilation, including

what the compiler performed to optimize the code, what optimizations were unsuccessful (and why), how data is accessible, relationships between functions, and so on. In program analysis, for example, type information has been used extensively. Aside from the input provided by the compiler, program analysis tools provide further information to the programmer [179, 180, 181]. This has also proven useful for bug detection and verification tools, as well as in IDEs to aid in program development.

Some of these tools perform *static analysis*, which involves looking at the code, either as text or as a syntax tree, and determining attributes such as module interdependency and uncaught exceptions. For example, in a control flow graph [182], each executable statement in the code can be represented as a node with edges connecting nodes that can be executed sequentially. Then, data flow markers are propagated around the control flow graph to collect all data flow, and, as we approach each node, we can search for unusual patterns that could indicate errors − perhaps the first such a tool to be widely used was Lint [183], which became a standard tool for C programmers.

Another example is *common subexpression elimination analysis* [5, 158], which identifies instances of identical expressions at some point in the program, and replaces them with a single variable holding the computed value. Another analysis that works in a similar manner is *alias analysis* [184]. In many languages, for example, an alias is defined as two or more expressions that denote the same memory location and is introduced either by pointers, call-by-reference, array indexing, or unions (due to shared memory) in C/C++. Alias information [185, 186] is important because it can improve the precision of analyses that require knowing what has been modified or referenced, removing redundant loads/stores and dead stores, identifying objects to be tracked in error detection tools, and parallelizing code (e.g., recursive calls to Quicksort can be made in parallel provided that each call refers to distinct regions of the array [187, 188, 189]).

Over time, compilers incorporate more and more static analysis features. The `g++` compiler, for example, has a number of collections "flags" that enable numerous warnings such as `-Wall`, `-Wextra`, `-pedantic`, and `-Weffc++` to name a few [190]. One concern is that these are frequently not enabled by default and must be enabled using command-line arguments. A few examples include: out of bound accesses to arrays (when compiled with `-o2`);

bad formats in `printf`; possibly uninitialized variables; use of `char` in array subscripts; potentially uninitialized variables; unused function parameters; empty loops; for classes with dynamic allocation, failing to implement your own copy constructor and assignment operator; and comparisons between signed and unsigned data types.

Since not all useful analysis can be done statically, there exists additional tools that perform *dynamic analysis* to examine the execution flow of a program. Conventionally, the C++ standard library does not check for typical misuses like overflowing an array or accessing elements that do not exist. In some respect, the `assert` command in C++ and Java can be thought of as the languages' own extension mechanism for such checks. Pointer errors in C++, which are normally unchecked by standard run-time systems, are also common [191]. Examples of ways to catch pointer errors include using *fence-posts*[23] [193] around allocated blocks of memory, adding tracking information to allocated blocks in order to indicate the location of the allocation calls, or adding a secondary bit to allocated blocks that is cleared when the block is first allocated and set when the block is freed[24].

In object-oriented languages, the optimization of method dispatch is another area where type information is helpful [195] (e.g., [196, 197, 198]). For example, a run-time lookup of the code associated with the method invoked on an object, followed by a costly indirect jump to that code, makes general method dispatch an expensive operation. If an object instance of a class is known at compile-time, a more efficient direct method invocation code can be generated instead. The method's code can even be expanded *in-line* at the moment of call if it is small enough. Furthermore, a simple analysis of the object's static type and the program's class structure − and hierarchy − reveals numerous chances for optimization. For example, if the type of an object is a class with no subclasses, then the compiler knows the *real* type of the object and, therefore, may create direct invocations for all methods of the object [199, 200].

Static and dynamic program analysis, as it turns out, are both aimed at automatically

[23] These are the areas immediately above or below memory allocations. The debug *malloc* library [192] can put special values in the areas surrounding each allocation so that it can detect when or if they have been overwritten. Note, the library does not notice when the program reads from these locations; only when it writes values does the library notice.

[24] There are currently free memory analysis tools like Memcheck, which is built on the Valgrind framework, and commercial tools like BoundsChecker, IBM Rational PurifyPlus, and Intel Inspector [194].

answering questions regarding the possible behaviors of programs. However, some of these analyses have unusual characteristics, primarily because they deal with static typing. In fact, static typing is what ensures that no type errors occur during program execution and allows the compiler to generate efficient code with a minimal memory footprint.

3.6 Type Inference

Even though type information is frequently provided in the source code, it can also be extracted directly from known types of individual symbols that occur in an expression. This is performed in order to determine the type argument (or arguments) that make an invocation applicable, the type of the result being assigned to a variable or being returned, the type of global variables, the type of literals, etc. For example, when a method is defined, type inference is used to determine the type arguments, return type, etc. When this is not possible, perhaps, due to a lack of external information, type inference is postponed until more information is available, such as at an invocation. This process is occasionally referred to as "incremental type inference" [201, See Section 3].

Example 1: Consider a generic class with a generic constructor like this

```
class A<T> { ❶
  <T> A(T t) { } ❷
}
```

and the following instantiation of the class G

```
A a = new A<Integer>(""); ❸
```

❷ contains a constructor with a formal type parameter `T` that is not the same as the formal parameter of the generic class `A<E>` ❶. Later in ❸, we specify the explicit parameter type *Integer* for the formal type parameter `E` of the generic class `A<E>`, and use the empty string (i.e., `""`) for the compiler to infer that the formal type parameter `T` is *String*. In general, we do not need to declare type parameters for generic method calls, such as

invocations to constructor, because the Java compiler can infer them [202, See Chapter 18]; that is, the compiler takes advantage of the *target type* of an expression (i.e., the type that the compiler expects) to infer type parameters. So, we may substitute the parameterized type of the constructor, similar to what we did for its argument, with an empty set of type parameters. For example, we could have also written ❸ as

```
A<Integer> a = new A<>("");
```

□

The purpose of type inference is to recreate the types of expressions using known types as a starting point. The distinction between type inference and compile-time checking is essentially a matter of degree, even when type checkers are used to conduct some of this. For example, a type checking algorithm runs through the program to ensure that the types declared by the programmer match the requirements of the language (e.g., [203, 204, 205]). Meanwhile, the concept underlying type inference is that certain information is not stated; therefore, some form of logical inference is needed to figure out what kinds of identifiers are being used (e.g., [94, 206, 207]). The difference between these two is depicted in the Figure 9. Note, the types that are crossed in (b) are the ones we should infer.

(a) `int foo(int a){ return a*2; }`

- **Type checking**
  - Examine the body of each function
  - Check language requirements by using declared types.

(b) ~~`int`~~ `bar(`~~`int`~~ `b) { return foo(b+2)/4; }`

- **Type inference**
  - Examine the code without type information
  - Determine the *most* general types that could have been declared.

Figure 9. Type checking vs. type inference.

Two common techniques to inference types in object-oriented languages are Hindley–Milner type system [84], and Palsberg and Schwartzbach [206]. All type infer-

ence methods based on the Hindley–Milner technique have the same goal: To infer principle types. The behavior of the type inference algorithm is briefly discussed next, with the goal of conveying some of the key concepts.

Let us start by looking at (b) in Figure 9, in particular, let us look at the expression `foo(b + 2) / 4`. Going from left to right, we can see that the type of the expression is *int*. How do we know this? Well, we can see that the type of the + operator is $int \times int \rightarrow int$. We also know that the literal 2 has type *int*, so we conclude that `b` takes the type of 2 (which is *int*), the type expected by the addition operator. This suggests that the return type of `foo` must also be *int* because the result type of the addition operation is *int*, and so we have $int \rightarrow int$. From here, it becomes a trivial exercise to determine the type of 4 and the return type of `bar`.

It should be fairly clear that Hindley–Milner's type inference technique determines types of definitions in order. That is, it infers later definitions by using the types of earlier definitions, and it collects and solves some constraints for each definition in order to establish the type of each (sub)expression. Figure 10 is an example that demonstrates how a typical type system with type inference would work. The HM algorithm [208, 209] would build a parse tree containing all of the elements, assign type variables[25] (t_i) to the tree's nodes and use the types that are already known (e.g., literals and operators), and then generate constraints[26] in order to determine the unknown type of each element.

[25] In this instance, *type variables* are special variable that serve as place holders for types that we do not yet know. They are eventually substituted with other types during type inference and thus disappear from the syntax tree.

[26] These are a means to keep track of what we learn about types and their relation between them, as we progress through the type inference process.

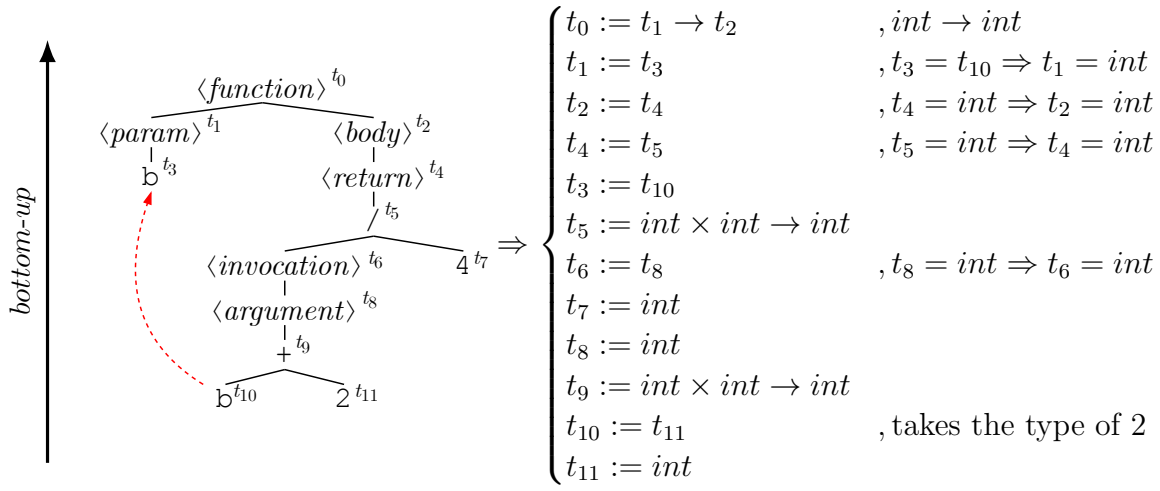


Figure 10. Type inference: collecting and solving constraints.

Figure 10 is an example of a bottom-up type inference, which is the kind where type information flows from the leaves of the tree up to the root [210, 211]. It is the most prevalent type inference method, and it is utilized in a lot of statically typed languages. However, type information can travel in two directions in some modern languages, such as Swift: top-down and bottom-up (also known as *bi-directional* type inference) [212]. This can be seen in literals, implicit member expressions, closures, and function call expressions, among other expression types.

Example 2: We have already seen how bottom-up inference type works. We will now demonstrate how top-down inference type works in Swift when using and declaring a few constants without having to specify any types at all, as the compiler can deduce that information from the values being assigned:

```
let var1 = 42; // var1 is of type Int ❶
let var2:Double = 42; // var2 is of type Double ❷
let var3 = [1,2,3,4] // var3 is of type [Int] ❸
let var4:Double = var1; // Error ❹
```

A literal value like the one in ❶ has type `Int` by default (so `var1` is of type `Int`), but we can change it to type `Double` with an explicit type annotation, as seen in ❷. This means that in a constant or variable declaration, the type annotation serves as a context

type for the initial value (i.e., what a value is expected to be). Thus, even when the literal 42 is of type `Int`, there is an implicit conversion from `Int`-to-`Double` in ❷; however, there is *not* an implicit conversion in ❹ because `var1` is an instance of type `Int`, so we end up with an error. □

It should be pointed out that Swift's bi-directional type inference is implemented via a constraint-based system; that is, of comparison to a traditional type system, it captures more information about the possible values in a variable.

Of course, while the syntax is not so much of importance, we may use a variable type annotation (as seen in Example 2) to override the inferred type of a variable in Figure 9; for example, we could have written

```
bar(b:int):int {...}
```

instead of

```
~~int~~ bar(~~int~~ b) {...}
```

3.7 Type Annotation

While a compiler can use *type inference* to type check a program that is completely free of types, a type annotation − also known as *type signature* − can be used to indicate the type of a variable or expression to the compiler and someone reading the code (i.e., it explicitly specifies the type of a variable or expression) [113, 114]. Compare to other forms of annotations, they are more precise than an annotation − or attribute − on a program element (often a class, method, or field in languages like Java, Groovy, and C#) or program comment, and they are easier to check. In Python, for example, a type annotation begins with a colon and ends with a type after declaring or initializing a variable or an expression as seen below [213]. For this example, we use ▬ to group type annotations.

```
def add(num1: int, num2: int) -> int:
 ...
```

For **dynamically typed languages**, this gives their typed system a sense of statically typed control; for example, in Python, a type can be annotated – with some optional *metadata* if needed – for static analysis tools like MyPy [214] and PyType [215] to type check a program. In order to enable type checking, Python introduced PEP 484[27], which specifies a syntax for optional type annotations [217]. A type checker, such as MyPy, provides a formal language for expressing types while also ensuring that the specified types match the implementation (and optionally that they exist).

Example 3: Consider the following Python code:

```
def fib(n: int) -> Iterator[int]:
 a, b = 0, 1
 while a < n:
   yield a
   a, b = b, a+b
```

When using MyPy, the type checker ensures that this function is invoked with the correct type arguments, or that the result of this function is assigned to a variable of the appropriate type before execution. For example,

```
Iterator[complex] it = fib(4)
```

will generate an error before the program executes because the expected type is of type `Iterator[int]`. However, adding *type hints* allows third-party IDEs (e.g., PyCharm [218]) and linters to flag (or highlight) erroneous annotations, acting as an early warning system that something is wrong with our logic. □

Thanks to type annotations, we can use static code analysis to catch type errors before the execution of a program. This is done automatically by PyCharm, which highlights type

[27] PEP 484 provides a number of ways for inspecting and verifying the types of objects in Python scripts, such as type annotations and type comments [216].

errors (albeit, they are often easy to miss), as demonstrated above. In this scenario, type hinting can assist with data validation and code quality, code speed (since some assumptions can be made), as well as code documentation and readability. Aside from static type checkers, dynamic type checkers can also ensure that the type of variables and return values match the declared type hints at run-time.

3.8 Types and Type Checking

We have indirectly talked about types and type checking, but we have not formally defined them yet. So, what exactly are types and how do they relate to type checking? Types play a crucial role in programming languages. For example, to arrive at a result − either explicitly or indirectly − every program utilizes data, which is often collected in the form of data structures and handled by algorithms. This means that types specify the actions that may be performed on the data, the meaning of the data, the methods for storing values of that type, and the values that a variable or a function can accept. Therefore, the applications of data types can be categorized into two groups: *checking* and *translation*.

- **Type checking:** With types, we can verify that valid operations are applied to variables and expressions, and raise errors for anything that might cause a type error. Type checking makes use of logical rules to reason about a program's behavior at run-time; for example, a function with a type $boolean \times boolean \rightarrow boolean$ expects its two arguments to be of type *boolean*; the result is also of type *boolean*.

- **Translation:** With types, we can determine the amount of storage required for a variable based on its type at compile-time or run-time. Local data storage can be allocated statically or managed through a block-structured language's standard stacking mechanism. These strategies are appropriate for tables carrying contextual information rather than the program text [24]. Such information is sometimes used in semantic analysis to identify operators and check construct compatibility with their environments; for example, calculating the address denoted by an array reference.

Formally, we can say that a *type* in a language represents a set of values and the operations

and relations that are applicable to them in order to ensure that meaningless operations, such as dividing the integer 7 by the string `"seven"`, are not performed. For example, the type `int` (typically, a 32-bit type) represents the set of whole numbers and the typical integer operations and relations, including `+`, `-`, `*`, `/`, `%`, `=`, `<`, `>`, to name a few. A type can be defined as follows:

Definition 3 (Type): A *type* is a set of values, $\mathcal{V}$, and a set of operations, $\mathcal{O}$, applicable to the values in $\mathcal{V}$ [5]. □

In other words, a type specifies the possible values of a structure (e.g., such as a variable), its semantic meaning (e.g., how a piece of data is supposed to behave; that is, `int x` in Java means $-2^{-31} \leq x \leq 2^{31} - 1$), and how its values can be kept in memory (e.g., an `int` in Java is a 32-bit signed 2's complement integer).

It is important to mention that the result of applying an operator to a type is not *always* guaranteed to yield a value from the type's set of possible values [5]; that is, for a type T with a set of values $\mathcal{V}$, a set of operators $\mathcal{O}$, an n-ary operator $o \in \mathcal{O}$ and n values $v_i \in \mathcal{V}$, there is no guarantee that $o(v_1, ..., v_n)$ is a value in $\mathcal{V}$. For example, if variables x and y are declared to have a type *integer*, a static type system will assign the expression $x < y$ the type *boolean*, which guarantees that the value of the expression at run-time will be boolean. Thus, a static type checking system detects when an expression is valid and should ensure that the value of a type checked expression is compatible with its static type.

Every single bit of data that is processed in a program is categorized into two types commonly known as **atomic types** (*primitives*) and **composite types** (*constructed types*); therefore, a type is either a composite type constructed by applying a type constructor (see Definition 4) to built-in types or a type from a collection of built-in types. However, despite the fact that these types serve quite different purposes and may be distinguished from each other, their concepts are frequently misunderstood in some languages. For example, some languages offer built-in support for different primitive types or types composed of more than one − primitive or constructed − type (e.g., such as a *string*[28]) than other languages.

[28] A *string* is an atomic type is some languages; for example, JavaScript.

Definition 4 (Type Constructor): A *type constructor* is a language construct that allows new constructed types to be created from existing atomic or other constructed types [5]. □

While different levels of type checking expressions can be applied to different areas of the source code (this is done in order to rule out undefined behaviors), the general idea is to first build a parse tree, assign a type to each leaf element, and then assign a type to each internal node with a post-order walk. In order to do type checking, a compiler must assign a *type expression* to each component of the source code so that we are able to represent types that are defined in a program [3]. After that, the compiler must check that these type expressions follow a set of logical rules known as the source language's type system. That is, we use them to ensure that the program satisfies both the general semantic rules and the specific ones concerning the language constructs. We define type checking as follows:

Definition 5 (Type Checking): A *type checker* is a program that verifies that operations are performed with the correct types in line with the type system specification of a language [3]. □

Type checking makes up a large part of the semantic analysis process. Generally, this signifies that all operands in a given expression are of the correct type and number. For statically typed languages, the type checker obtains the information it requires from declarations and stores it in a symbol table. Nevertheless, the rules for operations are sometimes established by other sections of the code (e.g., as in function prototypes), and other times they are defined as part of the language's definition (e.g., in a binary arithmetic operation, both operands must be compatible[29] or of the same type). For dynamically typed languages, on the other hand, type checking is done at run-time by providing type information for each data location. A variable of type *string*, for example, would include the actual string value as well as some sort of "tag" indicating that it is of type string. Naturally, any operation execution begins with a check of these tags, and the operation is only carried out if and only if everything checks out; otherwise, a type error occurs, which normally causes the program to halt.

[29] A compatible type is one that is either permissible for the operator or one that is allowed by the language rules to be implicitly converted to a legal type by compiler-generated code.

Although some languages are stricter about *coercion* than others, most − if not all − compilers have built-in capabilities for rectifying even the most basic of type errors, such as when a compiler finds a type error and then changes the type of the variable to an appropriate type. This happens in Java when an addition operation is performed on a mix of integer and floating point values (as seen in Section 3.3 on page 36). In fact, whenever a type error occurs in Java, the compiler looks for a suitable conversion operation to put into the generated code to correct the problem; thus, a type error occurs only if no conversion can be performed. The question of whether or not to provide a coercion functionality is debatable. Coercions can prevent a programmer from having to worry about minor details, but they can also hide sever flaws that would otherwise be discovered during compilation.

3.8.1 Implementation

Recall, the semantic analysis step is concerned with verifying language rules, particularly those that are too complicated or impossible to limit in the grammar. To give an idea, here are a few semantic rules from the Espresso type system:

Rules	**Description**
arrays	the index of an array must be of type integer
expressions	for a binary expression $e_1 + e_2$, both expressions must be of numeric type
methods	each actual parameter in a method call must have the same type as or be compatible with the type of the formal parameter
class	the parent of a class, if supplied, must be a correctly declared class type
interfaces	if a class declares that it implements an interface, it must implement all of the interface's methods

Even though semantic checking centers around types, as illustrated in the previous examples, we also need to verify that identifiers are not reused inside the same scope, that identifiers are declared or defined before they are used, etc. A type checker's implementation entails going over all of a language's established semantic rules and ensuring that they are followed consistently. The recording of type information for each identifier is the first step

in developing a type checker for a compiler. The name of the identifier is all a scanner understands, and this is what is sent to the parser. After parsing each identifier's declaration, the parser builds a declaration record for it to be stored later. When the semantic analyzer encounters uses of an identifier, it can search up that name and identify the matching declaration, or it can report if no declaration is found. This is a crucial step because it allows us to type check if the variable is used with an incompatible type or in a section of code where the type is illegal.

In a language like Espresso, for example, the representation of base types (i.e., atomic types) and array types is rather simple. Classes are more complicated because they must have a list or table of all fields − variables and methods − present in the class in order to allow access and type checking of the fields. Classes must also enable the inheritance of all parent fields, which can be accomplished by linking the parent's table to the child's or copying the parent table's contents to the child's. Interfaces resemble classes, but they simply include method prototypes and no implementation or instance variables (except for constant variables).

The last step in implementing a type checker is to create a list of the semantic rules that govern which types are allowed in which language constructs. There are also general guidelines that must be followed, rather than just a single construct, such as declaring all variables, making all classes global, and so forth.

3.8.2 Designing a Type Checker

When designing a type checker for a compiler, one common way to do this is to:

1. Determine which types are available in the language.
2. Determine which language constructs have types associated with them.
3. Determine the language's semantic rules.

A language's type system is defined by the combination of these three things: Types, appropriate constructs, and rules [3, 5]. We can use a type checker as part of the semantic analysis step of a compiler once we have established a type system. In Espresso, we have **base**

types (*int*, *short*, *boolean*, *double*, etc.) and **composite types** (*arrays*, *classes*, *interfaces*, etc.). Arrays of any type, including other arrays, can be created. **Complex types** – we may recognize these as abstract data types (ADTs), such as stacks, queues, linked-lists, hash-tables, etc., can be constructed using classes, but they are not treated any differently from classes, so we will not need to consider them (Espresso does not have support for these type of higher-level abstractions).

Static type checking in Espresso consists of two separate processes: inferring the type of each expression from the types of its components, and confirming that the types of expressions in certain contexts match what is expected. Since these two steps can be combined logically into one, we do so. How do we then determine the type of an expression? We can think of the process of identifying the type of an expression as **logical inference**. For example, if x is an identifier that refers to an object of type T, then expression x has type T. Moreover, if the operands E_1 and E_2 of $E_1 + E_2$ are known to have types *string* and *string*, then $E_1 + E_2$ has type *string*. On the other hand, a class type does not required any checking, but it is in itself a type, so we simply return this as a type. It is evident that all of this information can be encoded using inference rules provided that all parse tree nodes that are descendants of *Expression* return their type after the type checking has taken place. Simply put, type checking in a context is a collection of inference rules expressing yet another judgement (i.e., an assertion)[30], and these rules correspond closely to the recursive AST traversal that implements them.

Why specify types this way? It turns out that doing so provides a precise definition of types that is independent of any particular implementation. That is, there is no need for Espresso's compiler (or any compiler) to have the same rules as the reference Java compiler for example. A type checker can also be implemented in any way as long as we follow the rules, giving us the maximum flexibility possible in implementation. Just as importantly, it also allows formal verification of programs properties as we can do inductive proofs on the structure of programs (which is what it is used in the literature). In closing, a static type system makes it possible for a compiler to catch many common programming errors, but at

[30] Hence, type inference is nothing more than an attempt to demonstrate a different judgement by going backwards through the rules.

the expense of preventing some valid programs. While others advocate for more expressive static type checking, some advocate for dynamic type checking in its place (even when they are also more complex). In the next chapter, we will illustrate the process previously discussed in the context of Espresso to make it more tangible.

Chapter 4

The Espresso Language

4.1 Introduction

We define a Java subset, a safe language called Espresso, that replicates Java's most important features (without generics). Additionally, we define a type inference system to provide compile-time language checking and operational functions [31] to describe how a Espresso program behaves in terms of the behavior of its components. Espresso includes classes, instance variables and methods, inheritance of instance variables and methods, shadowing of instance variables and methods, interfaces, dynamic method binding, assignments, and the *null* value. The operational semantics is defined in the context of configurations and terms, where configurations are tuple terms and states, and the terms represent the part of the original program to be executed. Textual substitution is used to explain method calls. The type system is described as an inference system in which the types of expressions are determined based on the types of some of the symbols that appear in them.

In the following sections, we will specify the syntax and the type system of Espresso. Note that a Espresso program will type check with the Espresso type system (as specified below) if it will type check with the Java type system (as specified in the Java language specification). Thus, while Java's static type system removes common programming errors right from the start, its type system's information is *not* always enough to prevent a run-time error from occurring.

[31] These include *type predicates*, *type equality*, *type equivalence*, and *type assignment compatibility*.

4.2 What is Espresso?

The concept of types is inextricably linked to our intuition. We have an intuition that integers are distinct from booleans, which are different from characters, which are different from other types. These differences arise from behavior: We can push anything onto a stack, but not onto a character; we can extract a substring with a specified length from a location in a input string, but not from an integer; we can define a structure and behavior for a *class*, but not for a *type*; and so forth. In Java, a *type* is all about behavior: An object may belong to exactly one class, but it may have many types at run-time. For example, a string object can be any of the following types: *String*, *Comparable*, *CharSequence*, or *Object*, but it only has one class, *String*. Corresponding this analogy, Espresso works with data in the form of values or *objects*, which have **types**; that is, descriptions of intended behavior and possible values for their datum, similar to most other object-oriented programming languages.

Being a true subset of Java, the meaning of an Espresso program is given by its meaning as a Java program. An Espresso program, therefore, consists of variable types, instance and class variable types, parameter and result types for methods, and interfaces of classes. There are five[32] important kinds of types in Espresso [10, see Section 3.8.3]; these are:

- **Primitive types** – these include *byte*, *short*, *char*, *int*, *long*, *float*, *double*, *boolean*, and *void*.
- **Null types** – the type of value *null*.
- **Array types** – the type of arrays of values.
- **Class types** – the type of objects.
- **Interface types** – similar to a *class* type.

Integers, floating point numbers, characters, and booleans all fall within the category of primitive types. The size of the integer that each of the four types of integers (i.e., *byte*, *short*, *int*, and *long*) may represent varies. Similarly, the range of floating point numbers that the two forms of floating point types (i.e., *float* and *double*) may represent also varies.

[32] In reality, there is a sixth kind of type: *enumeration* types, which we will discuss later.

Espresso will not complain if we treat a small integer as if it were a larger one; this means that it will allows us to keep a *short* value in a variable declared *long* since every short integer fits within the space designated for a large integer. Espresso also offers two additional non-numeric data types: *char* that represents special symbols, and *boolean* that represents truth values (i.e., *true* or *false*).

There are *class types* and *subclassing* to consider. These are the ones that must be instantiated before they can be used as types for values (e.g., these are the types of object). An "empty" value is represented by a special *null* type and most operations with it result in run-time errors or exceptions (e.g., an uninitialized reference variable). In addition, for any non-*null* type T, an array of values of type T has type T`[]`. Any type T may be used. We define the *depth* and the *base* of an array type for convenience and say that the *depth* of an array type is simply its nesting depth, whereas the *base* is the number of square brackets (`[]`) removed. Formally, we defined the depth and base as follows: For a type T,

$$depth(T) \stackrel{\text{def}}{=} \begin{cases} depth(T') + 1 & \text{if } T = T'\texttt{[]}\text{, strip } \texttt{[]} \text{ off } T' \\ 0 & \text{if } T \in \{\textit{primitive}, \textit{reference value}\} \end{cases}$$

$$base(T) \stackrel{\text{def}}{=} \begin{cases} base(T') & \text{if } T = T'\texttt{[]}\text{, strip } \texttt{[]} \text{ off } T' \\ T & \text{otherwise} \end{cases}$$

This will become clear when addressing the type expression of an array in Section 4.4.2 on page 66. We will also see the importance of this during type checking the four parse tree nodes – **ArrayLiteral**, **ArrayType**, **NewArray**, and **ArrayAccessExpr** – that deal with arrays in Espresso.

Implicit type conversions are confined to safe upcasts with respect to subtyping; all other (unsafe) conversions must be done explicitly, using either a conversion function or an explicit cast. In the general sense, upcasting is allowed whenever there is an *is-a* relationship between two classes. Espresso uses nominal subtyping, which means that a subtyping relationship is established when one type is explicitly declared to be a subtype of another. Note that we will refer to subtyping as *substitutability* in this case. For example, if T_1 is a subtype of T_2 (denoted as $T_1 \prec_\tau T_2$), values of type T_1 can be safely used where values of type T_2 are expected. (By safe, we merely mean that we will not attempt to apply an operation to a value for which the operation is undefined.) The subtyping relation is explained later in the chapter.

Finally, for the majority of Espresso types, type safety (consistency between compile and run-time types) is checked statically at compile-time. However, the compile-time type of a value may vary in some cases based on the program's control-flow and data-flow.

4.3 Syntax Extension

The grammar below uses the following notation with respect to Java in EBNF format [219]:

- Nonterminal symbols are words written in *italic* font.
- Terminal symbols are written in **bold** font.
- Production is of the form $lhs = rhs$, where lhs is a nonterminal symbol and rhs is a sequence of nonterminal and terminal symbols.

The abstract syntax of Espresso programs is given in Figure Figure 11. This syntax gives a variation of the Espresso grammar. This variation defines the types of our explicitly-typed intermediate language. Here, x and y are in the set of *variables* (where x is for method arguments and y for locals); cn, fn, and mn are in the set of variables of classes, functions, and methods; and T is in the set of *types*. There are four kinds of types T: primitive types (A), class names (cn), *Object*, and arrays. The notation $\overline{x_i}$ implies a sequence of the form $x_1, ..., x_n$, where the index set can be constrained as follows: $\overline{x_i}^{i \in \mathcal{M}}$ and $\overline{x_i}^{i \neq k}$, for example, where $\mathcal{M}$ could be a set or list, and k a number. The index i is omitted if ambiguity does not arise. Instead of the conventional **extends** keyword that Espresso and Java source programs use, we use $\prec_{\mathcal{T}}$ to denote a class extension.

Generic fragment:

$$
\begin{array}{rcl}
P \in \text{program} & = & \overline{K} \\
K \in \text{class declaration} & = & \textbf{class}\ cn \prec_{\mathcal{T}} C\ \texttt{\{}\ \overline{F}\ \overline{M}\ \texttt{\}} \\
F \in \text{field declaration} & = & T\ fn\ \texttt{;} \\
M \in \text{method declaration} & = & T\ mn\ \texttt{(}\overline{T\ x}\texttt{)}\ \texttt{\{}\ S\ \textbf{return}\ E\ \texttt{;}\ \texttt{\}} \\
T \in \text{type} & = & C \mid A \mid T\texttt{[]} \\
C \in \text{class} & = & cn \mid \textbf{Object} \\
A \in \text{primitive type} & = & \textbf{byte} \mid \textbf{short} \mid \textbf{char} \mid \textbf{int} \mid \textbf{long} \mid \textbf{float} \mid \textbf{double} \\
 & \mid & \textbf{boolean} \mid \textbf{void}
\end{array}
$$

$$
\begin{array}{rcl}
S \in \text{statement} & = & \textbf{if}\ (\ E\ )\ S_1\ \textbf{else}\ S_2 \\
& \mid & \textbf{while}\ (\ E\ )\ S \\
& \mid & \textbf{return}\ E\ \textbf{;} \\
& \mid & \{\ \overline{S_i}\ \} \\
& \mid & E\ \textbf{;} \\
E \in \text{expression} & = & y \mid E.y \mid E_1 \odot E_2 \mid E_1 = E_2 \mid E(\overline{E}) \mid \textbf{new}\ cn \\
& \mid & E_1[\ E_2\ ] \mid (\ cn\ )\ E \\
\\
cn & \in & \text{class names} \\
fn & \in & \text{field names} \\
mn & \in & \text{method names} \\
x, y & \in & \text{variables}
\end{array}
$$

Figure 11. Abstract syntax of Espresso.

4.3.1 Overview of the Abstract Syntax

The syntax is very simple since we only have to model a few constructs of Espresso. In Espresso, a program consists of a main class and a list of class declarations $\overline{K}$. A class definition contains definitions for field identifiers and their types, method identifiers and their signatures, and introduces a new class as a subclass of another (when no explicit superclass is specified, the superclass is considered to be *Object*). The name of the method, the names and types of the arguments, and a statement sequence make up the method body. Each method body must contain exactly one return statement, and it must be the last statement. This simplifies the operational semantics without limiting the flexibility of a method definition (e.g., it only requires a simple change to the body of any Espresso method to satisfy this property). Only conditional statements, repetition (e.g., while loops), assignments, and method calls are to be considered valid statements in the program. Values (which come in the form of primitives, references, and arrays), method calls, and access to instance variables are also taken into account. Interfaces are not included in this summary due to space constraints, although they can easily be added.

As Espresso statements we have conditionals **if** (E) S_1 **else** S_2, loop statements **while** (E) S, method returns **return** E**;**, statement composition $\{\ \overline{S_i}\ \}$, and expression statements E**;**. As Espresso expressions we have identifiers y, selections $E.y$, binary expressions $E_1 \odot E_2$,

assignments $E_1 = E_2$, method applications $E\,(\,E_1, ..., E_n)$, new object creations **new** cn, and array access $E_1[\,E_2]$. For simplicity we leave out method types of the form $(\overline{T}) \rightarrow T'$, and parameterized classes of the form $C\texttt{<}\overline{A}\texttt{>}$, and only focus on the ones previously described. We shall now proceed to an extensive and thorough explanation of the type system, type environment, and types of Espresso. We will return to its static type checking rules in Section 4.4.6 on page 92.

4.4 Type System

A typing judgment is commonly used to express the declarative specification of a type system.

$$\Gamma \vdash e : \tau$$

In a type system like Espresso's, however, Γ may contain classes, subclasses, and interfaces, as well as class hierarchies. It may also contain the type definitions of variables and methods of classes and interfaces. The standard way to express these are with the judgments

$$\begin{aligned} & \vdash P \\ & \vdash mc \\ & \vdash K \\ C & \vdash F \\ C & \vdash M \\ \mathbf{\Lambda}, C & \vdash S \\ \mathbf{\Lambda}, C & \vdash E : T \end{aligned}$$

meaning, the judgment $\vdash P$ means the program P type checks. The judgment $\vdash mc$ means the *main class* type checks. The judgment $\vdash K$ means that the class declaration K type checks. The judgment $C \vdash F$ means that the field declaration F type checks if defined in class C. The judgment $C \vdash M$ means that the method declaration M type checks if defined in class C. The judgment $\mathbf{\Lambda}, C \vdash s$ means that the statement S type checks if defined in class C, in a type environment $\mathbf{\Lambda}$. The judgment $\mathbf{\Lambda}, C \vdash E : T$ means that the expression E has type T if defined in class C, in a type environment $\mathbf{\Lambda}$. As we shall see later, we can think of syntax analysis as proving claims about the types of expressions. We begin with a set of axioms, then apply our inference rules to determine the types of expressions to validate statements in the program.

4.4.1 Type Environment

A type environment is a set of identifiers that are mapped to types in a finite way. In Espresso, we will approach environments a little loosely, considering them as sequences of bindings at times and sets of bindings at other times. We will also make some assumptions, such the fact that each variable can only exist once in a given environment. We use $\mathbf{\Lambda}$[33] (short for *environment*) to range over type environments; we use $Dom(\mathbf{\Lambda})$ to represent the domain of $\mathbf{\Lambda}$. If $var_1, ..., var_n$ are pairwise distinct variables (or identifiers), then the notation $\{\!\{var_1 : \tau_1, ..., var_n : \tau_n\}\!\}$ represent a type environment that maps var_i to τ_i for $i \in 1..n$; thus, $Dom(\mathbf{\Lambda}) = \{var_1, ..., var_n\}$ and $\mathbf{\Lambda}_{\boldsymbol{\tau}}(var_i) = \tau_i$; in other words, $\mathbf{\Lambda}_{\boldsymbol{\tau}}$ strips *var* off a type.

Types are now proven relative to the scope they are in, and so we write

$$\mathbf{\Lambda} \vdash e : \tau$$

if, in environment $\mathbf{\Lambda}$, the expression e has type τ. In pseudocode, we can infer the type of a variable using a *lookup* function; for example:

$$\mathbf{\Lambda}_{\boldsymbol{\tau}}(var_i) = \tau_i \equiv \begin{cases} \textbf{infer}(var_i, \mathbf{\Lambda}) \\ \quad \tau_i = \texttt{lookup}(var_i, \mathbf{\Lambda}) \\ \quad \textbf{return } \tau_i \end{cases}$$

The environment keeps track of all we need to know about each declared variable (as well as other named entities such as classes, methods, etc.) in the program. Although this works, we still need to improve the concept of context as it relates to blocks. Instead of a simple `lookup` table, $\mathbf{\Lambda}$ must be a "stack of `lookup` tables"; we separate the tables with dots as follows:

$$\mathbf{\Lambda}_1 \cdot \mathbf{\Lambda}_2$$

where $\mathbf{\Lambda}_1$ is an old (or outer) context, and $\mathbf{\Lambda}_2$ is a new (or inner) context, making $\mathbf{\Lambda}_2$ the innermost context; that is, $\mathbf{\Lambda}_2$ is the top of the stack.

$$\underbrace{\mathbf{\Lambda}_1 \cdot ... \cdot \mathbf{\Lambda}_n}_{\text{stack}} \equiv \begin{array}{|c|} \hline \varLambda_n \\ \hline ... \\ \hline \varLambda_1 \\ \hline \end{array} \leftarrow top$$

[33] $\mathbf{\Lambda}$ denotes a symbol table in Espresso.

With a stack of contexts, we start by looking in the top-most context and go deeper in the stack only if we do not find what we are looking for. Therefore, if $\mathbf{\Lambda}_1$ and $\mathbf{\Lambda}_2$ are type environments, then $\mathbf{\Lambda}_1 \cdot \mathbf{\Lambda}_2$ is a type environment defined in the following way

$$(\mathbf{\Lambda}_1 \cdot \mathbf{\Lambda}_2)(var) \stackrel{\text{def}}{=} \begin{cases} \mathbf{\Lambda}_2(var) & \text{if } var \in Dom(\mathbf{\Lambda}_2) \\ \mathbf{\Lambda}_1(var) & \text{otherwise} \end{cases}$$

where Λ_2, being at the top of the stack, takes precedence over Λ_1.

4.4.2 Types

It is not always necessary to know an expression's exact value; in many cases, knowing that an expression belongs to a broad class (or group) of expressions with similar properties is sufficient. If T denotes a type, an expression is of that type if it belongs to the class that T denotes. For example, in Espresso, if an expression corresponds to the class of expressions designated by the word *int*, which is the set of negative and positive numbers, it is said to be of type *int*. There are names for many of these types (classes), such as *integer*, *float*, *double*, etc. Other kinds are described by constructing more intricate type expressions from more primitive type expressions using the grammar (a set of rules for valid combinations). Types can also be used to describe expressions with the same underlying data structure, like arrays, list, hash-tables, etc., and can be categorized according to the way in which they are defined in the language.

In view of this, the concept of types must be introduced, including, where suitable, the possible values that a variable of a certain type can take. Although we will go over all of the constructed types one by one, we should keep in mind that some of these types − such as *record* [34], *union*, *procedure*, *pointer*, and *named type* − are not part of Espresso. However, it is important to mention them briefly as they are highly common from language to language. For practical purposes, the characters t and t_i are types; n and n_i are names; v_i is value; s is superclass; c_i is class, and α is type variable (only where noted).

Let us become acquainted with the principles of **type equality**, **type equivalence**, and **assignment compatibility** as described by [5] before going into any concrete types. First,

[34] In this thesis, a *Record* type is employed as a small use-case study in Chapter 5.

let us defined the operators $=_{\mathcal{T}}$ (type equal), $\sim_{\mathcal{T}}$ (type equivalent), and $:=_{\mathcal{T}}$ (assignment compatible) as follows:

Two types T_1 and T_2 are said to be **equal** if they are the same type. We can write

$$T_1 =_{\mathcal{T}} T_2$$

Given two types T_1 and T_2, we say that they are **type equivalent** if any value of type T_1 can be assigned to a variable of type T_2, and any value of type T_2 can be assigned to a variable of type T_1. We can write

$$T_1 \sim_{\mathcal{T}} T_2$$

Note that if $T_1 =_{\mathcal{T}} T_2 \Rightarrow T_1 \sim_{\mathcal{T}} T_2$, that is, if the two types are the *same* type (i.e., they are type equal), then they are naturally type equivalent. It should also be clear that $T_1 \sim_{\mathcal{T}} T_2 \Rightarrow T_2 \sim_{\mathcal{T}} T_1$. Finally, consider an assignment of the form

$$v = e$$

where v represents a variable and e expression. If we assume that the type of v is T_v and the type of e is T_e, and if a value of type T_e can be assigned to a variable of type T_v, then we say that T_e is **assignment compatible** with T_v. We can write

$$T_v :=_{\mathcal{T}} T_e$$

The compiler will locate the declared type of v (i.e., $\mathcal{T}(v)$), compute the static type of e (i.e., $\mathcal{T}(e)$), and determine whether the assignment is safe, impossible, or possible when type checking an assignment $v = e$, for example. (We write $\mathcal{T}(v) = T_v$, where $\mathcal{T}(\,)$ is a type computation function.) For the assignment to be *safe*, a variable must only contain values of its type, which is always true when the assignment is executed **if** $\mathcal{T}(v)_{\mathcal{V}} \supseteq \mathcal{T}(e)_{\mathcal{V}}$[35]. Otherwise, the compiler would have to deem the assignment as incorrect − and therefore *impossible* − or conduct a little further analysis. The further examination will then decide whether $\mathcal{T}(v)_{\mathcal{V}} \cap \mathcal{T}(e)_{\mathcal{V}}$ is $\emptyset$ or not. If they are disjoint, then the value e will never have the type of v; but if they are *not*, then it is *possible* for e to have the type of v. The three kinds of type checking scenarios are depicted in Figure 12.

[35] Recall, $\mathcal{T}(v)$ is a type and a type T is a value set $\mathcal{V}$, and a set of operations $\mathcal{O}$; thus, $\mathcal{T}(v)_{\mathcal{V}}$ is a set of types.

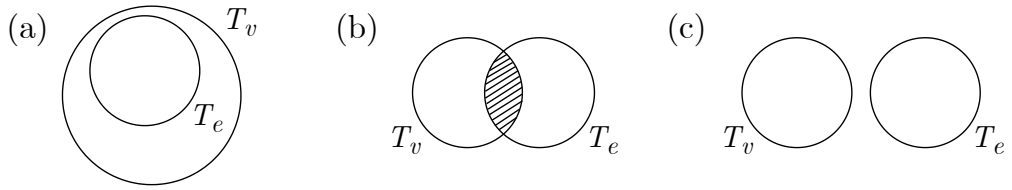


Figure 12. The three kinds of type checking scenarios: (a) safe, (b) possible, and (c) impossible.

Remark 2 ($:\neq_\tau$): It is important to highlight that $T_v :=_\tau T_e$ and $T_e = T_v$ is *not* the same [5]. The former means that a value of type T_e can be assigned to a variable of type T_v, and the latter that the two types are the same. Hence, the relation defined by $:=_\tau$ is not symmetric, meaning, $T_1 :=_\tau T_2 \not\Rightarrow T_2 :=_\tau T_1$; however, if $T_1 =_\tau T_2 \Rightarrow (T_1 :=_\tau T_2) \wedge (T_2 :=_\tau T_1)$. □

The remainder of this section will cover the representation for atomic and constructed types, which is analogous to type constructors in languages like C/C++ and Java. It is worth mentioning that, because type expressions are used to specify the type of a language construct, they will be used to link each Espresso language construct to a type expression. We also use types that are *not* part of Espresso. Again, we only include them here to demonstrated how they can be implemented in *Jiapi* as they are often used in many mainstream languages.

Even though we have just briefly addressed types, let us go over them again and this time talk about how each can be represented in a language. After all, types are away of grouping values based on the behavior we would like them to have.

4.4.2.1 Basic Types

An atomic data type such as *byte*, *short*, *char*, *int*, *long*, *float*, *double*, *boolean*, and *void* are type expressions. Depending on the language, a *string* could be considered a primitive or as a class, but in Espresso, it is a primitive type

4.4.2.2 Arrays

As is customary, an array can be defined as a contiguous collection of similar-type locations with an indexing mechanism that translates n integers into these locations. The dimension of an array determines the ranges in which the indices can change; generally, the first element in an array is at index 0, and the last at index $n - 1$, where n is the length of the array. How many bytes each location takes and how the bytes are interpreted is defined by the type associated with the array. An expression for constructing an array can be defined as

$$Array(baseType, [low..high])$$

We can construct an array where *baseType* is the type, and *low* is the lowest legal index, and *high* is the highest legal index. For example, the following snippet of code is a declaration of a fixed-length array in C, which allocates an array of size 5 at compile-time if and only if `a` is a global variable.

```
int a[5];
```

This array has the following type:

$$Array(int, [0..5])$$

In languages like Espresso, however, where an array is a heap-dynamic variable that is allocated by the programmer using the keyword `new`, an index set cannot be determined at compile-time. For example, the following also is a legal declaration of an array:

```
int a[] = new int[10];
```

This array has the following type expression:

$$Array(int, [\,])$$

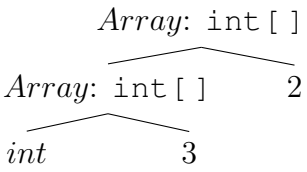
Note that in the event that the index set is unknown, we can simply use `[ ]` as the index set. For arrays with more than one dimension and fixed-length, like for example:

```
int a[2][3];
```

the type expression can be read as "array of 2 arrays of 3 integers each" and be written as

$$Array(Array(int, [0..3]), [0..2])$$

The internal representation of array `a` is depicted below as an abstract syntax tree.



Before we go any further, it is worth noting that some languages (such as C/C++, C#, and Java) do **not** have multi-dimensional arrays. Consider the two-by-three array defined below.

```
int a[][] = new int[2][3];
```

In a true two-dimensional array, all of the array's items occupy a continuous block of memory. In this situation, the *baseType* acts as a place holder for an real type (i.e., `int[]`) and, therefore, the type component of the entry for array `a` contains all the values needed to compute the address of any given element of that array. The type expression of this array is therefore:

$$Array(Array(int, [\,][\,])$$

This simply says that we should be able to form the type `base_type[]` for any *baseType* (i.e., an array of *baseType*). Thus, if we use `int[]` as the *baseType*, the type we should expect for array `a` is `int[][]`, or an "array of n arrays of m integers each", and that is exactly what happens in Espresso. Since the elements in a two-dimensional array of type `int[][]` are variables of type `int[]`, a variable of type `int[]` can only hold a reference to a one-dimensional array of type `int`. Basically, a two-dimensional array is just an array of *references/pointers*, each of which can refer to a single-dimensional, so these empty index

sets must be taken into consideration when determining type equivalence and assignment compatibility of arrays.

In order to create an array type with the appropriate dimensions in Espresso (the dimensions of the type plus the number of brackets on either the type or the name), we use the *depth*() function from Section 4.2 on page 59. For example, for

```
int a[][] = ...
```

the dimension is computed as follows:

$$\begin{aligned} depth(int\texttt{[][]}) &= depth(int\texttt{[]}) + 1 && \Rightarrow \text{strip } \texttt{[]} \text{ off } \texttt{int[][]} \\ &= \overbrace{depth(int) + 1}^{} + 1 && \Rightarrow \text{strip } \texttt{[]} \text{ off } \texttt{int[]} \\ &= \overbrace{0}^{} + 1 + 1 && \Rightarrow 0 \text{ because } T \text{ is } \texttt{int} \\ &= 2 && \Rightarrow \# \text{ of brackets} \end{aligned}$$

Remark 3 (Type Variables): Despite the fact that Espresso lacks generics, we feel compelled to discuss types that can serve as place holders for other types. A *type variable* is defined to be a variable that holds a value, but the value is now the set of possible types in the type system (i.e., atomic or constructed) [5], although there might theoretically be an endless number of constructed types. For example, in

$$Array(\alpha, [low..high])$$

α represent all possible array types. By assigning a type to the type variable α, and potentially a range as well, any specific may be obtained. □

4.4.2.3 Unions

Although a union resembles a record in terms of syntax, it can be defined as a value that can be one of several types. The basic representation for a union type is as follows:

$$Union(name, ((n_1, t_1), ..., (n_m, t_m)))$$

Here the only difference between a union and a record type is the word "Union" rather than "Record". This definition implies that a union can only include one member from its list of

members at any given moment. It also means that, regardless of the number of members in a union, only enough memory is used to store the largest member. Using the above representation, we can construct a union named *name* with fields n_i of type t_i for $i \in 1..m$; for example, in C++, the following snippet of code

```
union value {
  int i_val;
  double d_val;
};
```

means either `i_val` or `d_val`; that is, a variable of type `value` will hold either an *integer* value or a *double* value. The *value* union has the following type

$$Union(value, ((i\_val, int), (d\_val, double)))$$

4.4.2.4 Enumerations

This is a special data type that allows for a variable to be a set of predefined constant variables. The related constant values of enum members are − often in some languages and by default in others − of type *int*, starting at zero and increasing by one in the sequence of the declaration text. To define an appropriate type for an enumeration, we can just use

$$Enum(name, (v_1, ..., v_m))$$

to construct an enum named *name* with constant values v_i of type *int* for $i \in 1..m$; for example, in Espresso, the following code snippet

```
enum season {
  spring,
  summer,
  autum,
  winter
}
```

represents a choice from a set of mutually exclusive values. The method for accessing the values of an enumeration type varies per language. In C, we can get a value of an enumerated

type by just typing the element's name, whereas in C#, they are access much like fields in structs. The *season* enum, therefore, has the following type:

$$Enum(season, (spring, summer, autum, winter))$$

4.4.2.5 Procedures

Most languages do not consider procedures to be a type, but others do, such as ProcessJ, which has mobile procedures [220]. Typically, procedures[36] declare a return value type (or *void* if they do not return a value), a comma-delimited list of input parameters, proceeded by their data types, enclosed by round brackets, and a body enclosed between curly brackets. We can define a procedure type as

$$Procedure(name, ((n_1, t_1), ..., (n_m, t_m)), t)$$

and construct a procedure named *name* with a return type t and optional parameters n_i of type t_i for $i \in 1..m$; for example, the C++ function declaration bellow (with a pointer as an input argument)

```
void foo(int i, float f, double* d)
```

has the following type

$$Procedure(foo, ((i, int), (f, float), (d, \top)), void)$$

However, what should the type constructor for the unknown ($\top$)[37] be? It makes reasonable sense to define a pointer type and a type constructor for it, so that is what we will do next.

Note that we will not add the names of the parameters in the type because most languages (Espresso included) do **not** allow the same procedure to be re-implemented with the same name and type signature − parameters and return type; for type checking, we will use this instead

$$Procedure(name, ((t_1, ..., t_m), t)$$

[36] In this thesis, the term *procedure* is used as a synonym for method/function.
[37] $\top \equiv anything$.

4.4.2.6 Pointers

These are integral values that represent locations in memory; therefore, the value of a pointer (or reference) is a positive integral value. We can define a pointer type as

$$Pointer(\alpha)$$

where α is some type (i.e., atomic or constructed.).

Going back to the previous type constructor, we know that the type of d is in fact a pointer to a double. Recall, the $*$ operator takes a pointer to an object and returns the object, so d must be a pointer to an object of an unknown type α. If the type of d is represented by β, then $\beta = pointer(\alpha)$, and thus we say that the expression $*d$ has type α. We can now complete the type of *foo* to looks like this

$$Procedure(foo, ((i, int), (f, float), (d, Pointer(double))), void)$$

where

$$Pointer(double) = double*$$

Remark 4: Despite their resemblance to *pointers*, *reference variables* are intended to aid the compiler and programmer [221]. A reference variable is simply a pointer-sized chunk of memory that holds a pointer to the object. This variable, however, cannot be utilized in the same way that a pointer variable may. For example, in languages like C/C++, C#, and others alike, a pointer can be indexed like an array, but a reference cannot. A reference is tracked by the garbage collector in C#, while a pointer is not. A pointer can be reallocated in C++, but a reference cannot, etc. [26, 222]. □

4.4.2.7 Classes

In practice, we often need to create objects of the same kind in object-oriented language like Java and C++, and Espresso is no exception. That is because classes are a fundamental part of this language, so the ability to define classes is also a type constructor since defining a class is the same as constructing a type. A class consist of a piece of information annotated

with types at various positions (e.g., declarations of fields, such as variables and methods). It may also subclass another class or implement several classes. We can define a class type as

$$Class(name, s, (c_1, ..., c_m))$$

and construct a class named *name* with an optional superclass s, and an optional number of classes (or interfaces) c_i for $i \in 1..m$. Note that for a language with multiple inheritance like C++, it may be as

$$Class(name, (s_1, ..., s_m))$$

Consider the following Espresso code:

```
class A extends B implements C, D, E {
  ...
}
```

What is the type of an object of class A? As indicated before, an object may belong to exactly one class, but it may have many types at run-time. From this, we could say that it could be any one of these: A, B, C, D, or E (including *Object*); thus, the appropriate type of the expression `new A()` ought to be

$$Class(A, Class(B, , ()), (Class(C, , ()), Class(D, , ()), Class(E, , ())))$$

For brevity, it is understood that Espresso is a language with single inheritance. Multiple inheritance is possible in the language, but only through interfaces, so we will not discriminate the difference between a class and an interface; instead, we will use the same notation for an interface, except where noted.

4.4.2.8 Named Types

Last but important, a named type is commonly any explicitly constructed type (or a define type) that we create and give a name, such as a *class*, *struct*, *enum*, etc. In some languages, there is a type definition operator that can introduce aliasing for types, such as the

`typedef` operator in C/C++, which provides a "new type" by wrapping an existing type. A *NamedType* can therefore be used temporarily until it is resolved to a real type, often during name resolution.

A named type can be defined as

$$NamedType(name)$$

where *name* is used to create an additional name for another type.

Table 1 summarizes the primitive types and the type constructors we have just discussed.

Table 1. Atomic and constructed types.

Atomic Types	
Name	**Representation**
byte	*integer*
short	*short*
...	...
boolean	*boolean*
void	*void*
Constructed Types	
Name	**Representation**
array	$Array(\alpha, [low..high])$
record	$Record(n, ((n_1, t_1), ..., (n_m, t_m)))$
union	$Union(n, ((n_1, t_1), ..., (n_m, t_m)))$
enum	$Enum(n, (v_1, ..., v_n))$
procedure	$Procedure(n, (t_1, ..., t_m), t)$
pointer	$Pointer(\alpha)$
class	$Class(n, s, (c_1, ..., c_m))$
named type	$NamedType(n)$

Before we dive deeper into **type equality**, **type equivalence**, and **assignment compatibility** for each type in Espresso, we will introduce some type predicates that will be useful when we arrive at type checking.

4.4.3 Type Predicates

A type predicate is a method for defining a set of criteria and determining whether or not the provided object fulfills them. There is a type predicate for each type in Espresso; for example, $byte_?$, $short_?$, ..., $Class_?$. Here are some examples,

$$byte_?(\tau) \stackrel{\text{def}}{=} \begin{cases} true & \text{if } \tau \text{ is } byte \\ false & \text{otherwise} \end{cases}$$

$$short_?(\tau) \stackrel{\text{def}}{=} \begin{cases} true & \text{if } \tau \text{ is } short \\ false & \text{otherwise} \end{cases}$$

$$\vdots$$

$$Class_?(\tau) \stackrel{\text{def}}{=} \begin{cases} true & \text{if } \tau \text{ is class } C \\ false & \text{otherwise} \end{cases}$$

Additionally, it can be used with the following predicates:

- $Integral_?(\tau) = (\tau \in \{byte, short, char, int, long\})$
- $Numeric_?(\tau) = (Integral_?(\tau) \vee \tau \in \{float, double\})$
- $Primitive_?(\tau) = (Numeric_?(\tau) \vee \tau \in \{boolean, void\})$

We will now establish a number of type-to-type relationships as well as several functions that operate on types in the sections that follow. This will, in turn, help us define a set of rules for writing a type checker for Espresso. Since a type checker's primary function is, of course, to check types, then comparing types (type equal or type equivalent), checking if a type is a specific type (predicates), and evaluating if assignments are type-safe (assignment compatibility) are all part of this process.

4.4.4 Primitive Types

For numeric types, we need to introduce rules for coercion. This means that we need to specify the type hierarchy (or type lattice). Given two primitive types α and β, if a variable of type β can hold any value of type α, then we say that a primitive type α is "type-wise less than" another primitive type β. This definition appears to be that of assignment compatibility (as we shall see later), and it is, but it must be defined as an ordering operator (or $<_\tau$).

4.4.4.1 The Ordering Operator ($<_\tau$)

Given two numeric types α and β, if α and β are not the same type then $\alpha <_\tau \beta$ if all possible values of α can be held in a variable of type β **without loss of precision**; however, there are some values of β that cannot be held in a variable of type α without loss of precision. If we consider the ordering of types below

$$byte <_\tau short <_\tau char <_\tau int <_\tau long$$

as well as

$$float <_\tau double$$

and also

$$int <_\tau float \wedge long <_\tau float <_\tau double$$

we obtain the values column of Table 2. We can see that an *integer* variable, which can hold values from -2^{31} to $2^{31} - 1$, is capable of holding all the values of the short type; therefore, we can say that $short <_\tau int$. It should be noted that the $<_\tau$ operator is also transitive, that is

$$(\alpha <_\tau \beta) \wedge (\beta <_\tau \delta) \Rightarrow \alpha <_\tau \delta.$$

The $<_\tau$ operator, in addition, trivially extends to the $\leq_\tau$ operator, and $>_\tau$ and $\geq_\tau$ can be defined in a similar manner.

Table 2. Type hierarchy for atomic types in Espresso.

$\alpha <_\tau \beta$	byte	short	char	int	long	float	double	boolean	void
byte	F	T	T	T	T	T	T	F	F
short	F	F	T	T	T	T	T	F	F
char	F	F	F	T	T	T	T	F	F
int	F	F	F	F	T	T	T	F	F
long	F	F	F	F	F	T	T	F	F
float	F	F	F	F	F	F	T	F	F
double	F	F	F	F	F	F	F	F	F
boolean	F	F	F	F	F	F	F	F	F
void	F	F	F	F	F	F	F	F	F

Definition 6 (Least Upper Bound): We can define a ceiling function for two numeric types since $<_\tau$ is defined for all conceivable combinations of numeric types [5]; therefore

$$\lceil \alpha, \beta \rceil_\tau \stackrel{\text{def}}{=} \begin{cases} \alpha & \text{if } \beta \leq_\tau \alpha \\ \beta & \text{if } \alpha <_\tau \beta \\ \perp & \text{otherwise} \end{cases}$$

the least upper bound LUB(α, β) is an upper bound T of α and β such that there is no other upper bound of these types. That is, T is always one of the types α or β. In some instances, constructing the least upper bound for more than two types is required, in which case the least upper bound operator LUB($T_1, ..., T_n$) is defined as LUB(T_1, LUB($T_2, ..., T_n$)). □

The hierarchy in Figure 13 is a partial ordered set with a *least upper bound* (LUB) for each pair of elements. Two operands are transformed to the LUB for many binary operators (all of the arithmetic ones we are familiar with, excluding exponentiation). Thus, combining a *short* and a *char* implies the conversion of both to *int*; however, to add a *byte* to a *float*, the *byte* must first be transformed to a *float* (the *float* remains *float* and is not converted.)

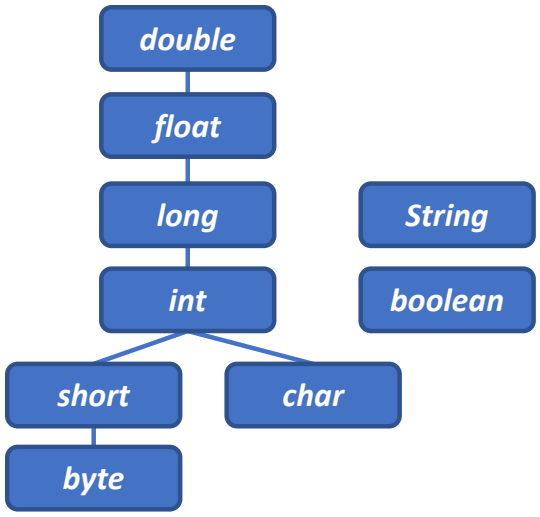


Figure 13. Espresso primitive type lattice.

Surprisingly enough, we obtain the same hierarchy as Figure 13 if we do not allow integral values to be assigned (without casting) to floating point variables; that is, $int \not\geq_\tau float$ and

$float \not\succ_{\mathcal{T}} int$ [5]. This means that an integer value cannot be assigned to a float variable, and vice versa − a float value cannot be assigned to an integer variable. The same goes for other non-comparable atomic types, such as *double* and *boolean*. A *boolean* value cannot be assigned to a *double* variable, and vice versa; then $boolean \not\succ_{\mathcal{T}} double$ and $double \not\succ_{\mathcal{T}} boolean$.

Because it is obvious that two atomic types with the same name are the same, we must define type equality next.

4.4.4.2 Type Equality ($=_{\mathcal{T}}$)

A primitive type α is only ever equal to another primitive type β if they are both the same type; that is

$$(\alpha =_{\mathcal{T}} \beta) \Leftrightarrow Primitive_?(\alpha) \wedge Primitive_?(\beta) \wedge (\alpha = \beta)$$

Note that the use of = means "the same" in the above formula.

4.4.4.3 Type Equivalence ($\sim_{\mathcal{T}}$)

Since the set of values that atomic types represent differ from type to type, type equivalence is the same as type equality. Thus, we can define $\sim_{\mathcal{T}}$ for two atomic types α and β as

$$(\alpha \sim_{\mathcal{T}} \beta) \Leftrightarrow Primitive_?(\alpha) \wedge Primitive_?(\beta) \wedge (\alpha =_{\mathcal{T}} \beta)$$

4.4.4.4 Type Assignment Compatibility ($:=_{\mathcal{T}}$)

Due to numeric type coercion and inheritance in Espresso, the $:=_{\mathcal{T}}$ operator is more than just type equivalence. For primitive types α and β, we have

$$(\alpha :=_{\mathcal{T}} \beta) \Leftrightarrow Primitive_?(\alpha) \wedge Primitive_?(\beta) \wedge (\beta \leq_{\mathcal{T}} \alpha)$$

Note that assignment compatibility is defined exactly as type equality for any remaining atomic types that are solely assignment compatible with themselves.

4.4.5 Constructed Types

For constructed types, the previous operators (and operations) may not always be as straightforward as they are for atomic types. This is mostly determined by the choices we make when defining the type system for a language. Starting with type equality, we will go over each operator once more.

4.4.5.1 Type Equality ($=_\tau$)

For constructed types that are named, we can use that name to determine if two types are the same. Of course, we assume that a name cannot be used twice, therefore there cannot be two types with the same name. Alternatively, we can use the structure that defines the types to determine if two types are the same.

Arrays

Two arrays $\alpha = Array(t_1, I_1)$ and $\beta = Array(t_2, I_2)$, where I_i represents the index set of each array, are equal if they are both arrays and both have type equal base types, or both are undefined ($\perp$)[38].

$$(\alpha =_\tau \beta) \Leftrightarrow Array_?(\alpha) \wedge Array_?(\beta) \wedge (t_1 =_\tau t_2) \wedge ((I_1 = I_2) \vee ((I_1 = \perp) \wedge (I_2 = \perp)))$$

Records

A record (also known as a *struct*) is a collection of named fields that are arranged in a specific order. Therefore, a record type's value is an ordered collection of values that are compatible with the record's types. Using *name equivalence*, we can say that for two record types α and β to be the same, they have to have the same type name, and so we get

$$\alpha = Record(name_1, ((n_{1,1}, t_{1,1}), ..., (n_{1,m_1}, t_{1,m_1})))$$
$$\beta = Record(name_2, ((n_{2,1}, t_{2,1}), ..., (n_{2,m_2}, t_{2,m_2})))$$

$$(\alpha =_\tau \beta) \Leftrightarrow Record_?(\alpha) \wedge Record_?(\beta) \wedge (name_1 = name_2)$$

[38] $\perp \equiv nothing$.

However, if we wanted our type checker to consider the structure of a record rather than just its name, we could instead write

$$(\alpha =_{\boldsymbol{\tau}} \beta) \Leftrightarrow Record_?(\alpha) \wedge Record_?(\beta) \wedge (m_1 = m_2) \wedge \big(\bigwedge_{i=1}^{m_1} t_{1,i} =_{\boldsymbol{\tau}} t_{2,i} \big)$$

which says that, in addition to our first definition, record types must have the same number of fields (i.e., $m_1 = m_2$), and the i^{th} field in α has to be type equivalent with the i^{th} field in β, and vice versa.

Unions

We can define type equality for union types in two ways, similar to how we defined record types. We can either check that two unions $\alpha = Union(name_1, ((n_{1,1}, t_{1,1}), ..., (n_{1,m_1}, t_{1,m_1})))$ and $\beta = Union(name_2, ((n_{2,1}, t_{2,1}), ..., (n_{2,m_2}, t_{2,m_2})))$ have the same type name,

$$(\alpha =_{\boldsymbol{\tau}} \beta) \Leftrightarrow Union_?(\alpha) \wedge Union_?(\beta) \wedge (name_1 = name_2)$$

or we can check if they have the same structure; therefore, for two union types

$$(\alpha =_{\boldsymbol{\tau}} \beta) \Leftrightarrow Union_?(\alpha) \wedge Union_?(\beta) \wedge (m_1 = m_2) \wedge \big(\bigwedge_{i=1}^{m_1} t_{1,i} =_{\boldsymbol{\tau}} t_{2,i} \big)$$

Enumerations

Along similar lines, two enums $\alpha = Enum(name_1, (v_{1,1}, ..., v_{1,m_1}))$ and $\beta = Enum(name_2, (v_{2,1}, ..., v_{2,m_2}))$ are equal if they have the same type name, such as

$$(\alpha =_{\boldsymbol{\tau}} \beta) \Leftrightarrow Enum_?(\alpha) \wedge Enum_?(\beta) \wedge (name_1 = name_2)$$

or, if they are structurally the same; therefore, we have

$$(\alpha =_{\boldsymbol{\tau}} \beta) \Leftrightarrow Enum_?(\alpha) \wedge Enum_?(\beta) \wedge (m_1 = m_2) \wedge (\{v_{1,1}, ..., v_{1,m_1}\} = \{v_{2,1}, ..., v_{2,m_2}\})$$

Procedures

For two procedure types to be equal, they must have the same name and the same signature; to that end, a procedure type is only ever equal to itself. For two types

$\alpha = Prodecure(name_1, (t_{1,1}, ..., t_{1,m_1}), t_1)$ and $\beta = Prodecure(name_2, (t_{2,1}, ..., t_{2,m_1}), t_2)$, we have

$$(\alpha =_{\mathcal{T}} \beta) \Leftrightarrow (name_1 = name_2) \wedge (m_1 = m_2) \wedge \Big(\bigwedge_{i=0}^{m_1} t_{1,i} =_{\mathcal{T}} t_{2,i} \Big) \wedge (t_1 = t_2)$$

Pointer

For two pointers $\alpha = Pointer(type_1)$ and $\beta = Pointer(type_2)$ to be equal, $type_1$ must be equal to $type_2$; then we have

$$(\alpha =_{\mathcal{T}} \beta) \Leftrightarrow Pointer_?(type_1) \wedge Pointer_?(type_2) \wedge (type_1 =_{\mathcal{T}} type_2)$$

Classes

We can define equality for two classes α and β by comparing their names; so, for $\alpha = Class(name_1)$ and $\beta = Class(name_2)$, we have

$$(\alpha =_{\mathcal{T}} \beta) \Leftrightarrow Class_?(name_1) \wedge Class_?(name_2) \wedge (name_1 = name_2)$$

Named Type

Finally, since all named types must resolve to real types, each named type must have a reference to its *actual type* in that two named types $\alpha = NamedType(n_1)$ and $\beta = NamedType(n_2)$ are only ever equal if their names and *actual types* are the same. Then, we have

$$(\alpha =_{\mathcal{T}} \beta) \Leftrightarrow NamedType_?(n_1) \ \wedge \ NamedType_?(n_2) \wedge \mathcal{T}(n_1) =_{\mathcal{T}} \mathcal{T}(n_2)$$

where $\mathcal{T}(n_i)$ resolves its actual type.

Table 3 summarizes the type equality, under name equivalence, for each constructed type [5]. For structural equivalence, we simply look at Table 4 on the next page and substitute $\sim_{\mathcal{T}}$ with $=_{\mathcal{T}}$.

Table 3. Type equality ($=_{\tau}$).

Type	Equality
record	name is the same
union	name is the same
array	base and index is the same
enum	name is the same
procedure	name and signature is the same
pointer	type is the same
class	name is the same
named type	actual type is the same

4.4.5.2 Type Equivalence ($\sim_{\tau}$)

Using the name to check for type equivalence involves determining whether two types have the same name; therefore, for all constructed types α and β, we have:

$$(\alpha \sim_{\tau} \beta) \Leftrightarrow (\alpha =_{\tau} \beta)$$

Checking structural equivalence, on the other hand, involves comparing the structure of types and their respective subtypes as seen before. Table 4 summarizes the type equivalence for each constructed type [5].

Table 4. Type equivalence ($\sim_{\tau}$).

Type	Equivalence
record	$Record_?(\alpha) \wedge Record_?(\beta) \wedge$ $(m_1 = m_2) \wedge (\bigwedge_{i=1}^{m_1} t_{1,i} \sim_{\tau} t_{2,i})$
union	$Union_?(\alpha) \wedge Union_?(\beta) \wedge$ $(m_1 = m_2) \wedge (\bigwedge_{i=1}^{m_1} t_{1,i} \sim_{\tau} t_{2,i})$
array	$Array_?(\alpha) \wedge Array_?(\beta) \wedge$ $(base\_type_1 \sim_{\tau} base\_type_2) \wedge ((I_1 = I_2) \vee$ $((I_1 = \perp) \wedge (I_2 = \perp)))$
enum	$Enum_?(\alpha) \wedge Enum_?(\beta) \wedge$ $(m_1 = m_2) \wedge (\bigwedge_{i=1}^{m_1} v_{1,i} \sim_{\tau} v_{2,i})$
procedure	$(name_1 = name_2) \wedge (m_1 = m_2) \wedge$ $(\bigwedge_{i=1}^{m_1} t_{1,i} \sim_{\tau} t_{2,i}) \wedge (t_1 \sim_{\tau} t_2)$

pointer	$Pointer_?(\alpha) \wedge Pointer_?(\beta) \wedge \alpha \sim_{\boldsymbol{\tau}} \beta$
class	$Class_?(\alpha) \wedge Class_?(\beta) \wedge (\alpha =_{\boldsymbol{\tau}} \beta)$
named type	$NamedType_?(\alpha) \wedge NamedType_?(\beta) \wedge$ $\rho(\alpha) \sim_{\boldsymbol{\tau}} \rho(\beta)$

4.4.5.3 Assignment Compatibility ($:=_{\tau}$)

The understanding of type equivalence is often determined on whether the language uses name equivalence or structural equivalence. In general, type equivalence appears to be an acceptable criteria for assignment compatibility given that a value of type T_e and a variable of type T_v are compatible if T_e can be assigned to T_v. Therefore, and with the exception of classes and arrays, for all constructed types α and β, we have

$$(\alpha_v :=_{\boldsymbol{\tau}} \beta_e) \Leftrightarrow (\alpha_v \sim_{\boldsymbol{\tau}} \beta_e)$$

where α_v is the type of the left-hand side variable v, and β_e is the type of the right-hand side expression e.

Arrays

Arrays are not quite as straightforward in Espresso. For two arrays α and β, a reference to α can be assigned to β if both α and β contain elements of the same type, or if both contain references and the type of reference contained in the elements of α can be assigned to the type of reference contained in β. But what if the array type holds a reference to an empty array − specifically if we have an assignment of the form $e = \{\}$? Writing a predicate function, like $ALAC$ (short for ArrayLiteralAssignmentCompatible), to determine if an array literal $e = \{e_1, ..., e_n\}$ can be assigned to an array type $T = Array(base\_type, [\,])$ can solve this problem [5]; so we have

$$ALAC(T, e) \stackrel{\text{def}}{=} Array_?(T) \wedge \begin{cases} true & \text{if } e = \{\,\} \\ \bigwedge_{i=0}^{n} \big(ALAC(baseType, \mathcal{T}(e_i))\big) & \text{if } e \neq \{\,\} \end{cases}$$

Example 4: The code below demonstrates how the array assignment compatibility rules are applied.

```
class B {...}
class A extends B {...}
...
int[] i = new int[4];
int j[];
short s[];
A[] a;
B[] b;
j = i;  // OK
s = i;  // Error
a = b;  // Error
b = a;  // OK
```

Because both variables are declared as references to arrays with integer values, assigning `i` to `j` does not generate an error. However, assigning `i` to `s` is incorrect because the variables are declared as references to arrays that contain different kinds of elements, none of which are object references. What about `a = b` and `b = a`? Before we answer this question, we need to remember a couple of things about classes in object-oriented languages. □

Now that we have defined what constitutes types in Espresso − and other common languages, we move on to answering the question: Under what circumstances a type α is a subtype of a type β; denoted as $\alpha \prec_{\mathcal{T}} \beta$? If we have an expression e_1 with type α and an expression e_2 with type β, we can sometimes safely use e_1 instead of e_2. As an analogy, the C function *getchar()* reads a character from *stdin* (the standard input), regardless of which standard input is used. It is almost as if we have an IO_Stream class with numerous derivatives, such as keyboard or disk, on which we may use *getchar*. In this case, the types keyboard and disk are both *subtypes* of IO_Stream; as an alternative, IO_Stream may be considered a *supertype* of keyboard and disk. It should be clear that a polymorphic type system is one that allows functions to increase generality by taking use of the possibility that a value may exist in several types [223].

Subtype polymorphism is the kind of polymorphism that we find in Espresso. It is based on the principle that there may be a relationship between types [32, 108, 223], and we use the $\prec_{\mathcal{T}}$[39] operator to indicate this. When using inheritance in Espresso, subtyping is performed

[39] Most literature uses $\sqsubseteq$, <:, or $\leq$ to denote subtyping, but we will stick with $\prec_{\mathcal{T}}$.

automatically; however, this does not imply that subtyping and inheritance are the same thing [32, 223]. There can also be instances of subtyping that are not inherited; this can be done with **interfaces**. A interface is a declaration that includes a list of method names − but not method bodies − and sometimes some constants. Hence, there is nothing to inherit because there is no code in an interface except for a (sub)set of methods. (We use the term "concrete" to denote that a class is *completely* defined, that is, it has all of the actual code for the methods.)

We define subtyping as follows:

Definition 7 (Subtyping): Given two types α and β, we define $\alpha \prec_{\mathcal{T}} \beta$ if and only if *whenever* the context requires an element of type β, anything of type α can be used. In other words, a type α is a subtype of a type β if it any object that belongs to α will also belong to β; therefore, it is legal to assign an instance of α var_1 to a reference of β var_2 such that $var_2 \leftarrow var_1$.

The $\prec_{\mathcal{T}}$ operator satisfies:

- If $(\alpha \prec_{\mathcal{T}} \beta) \wedge (\beta \prec_{\mathcal{T}} \delta) \Rightarrow \alpha \prec_{\mathcal{T}} \delta$
- If $(\alpha \prec_{\mathcal{T}} \beta) \wedge (\alpha' \prec_{\mathcal{T}} \beta) \Rightarrow (\alpha \wedge \alpha') \prec_{\mathcal{T}} \beta$
- If $\alpha \prec_{\mathcal{T}} \alpha \Rightarrow \alpha$[40]

□

Note that Definition 7 can alternatively be interpreted in terms of subsets as well, where, for example, if α is a subtype of β, then all elements of type α are automatically elements of type β. According to Definition 7, we say that A is a subtype of B, in Espresso, if one of the following requirements is *true*:

(1) A and B are the same type.

(2) A and B are both class types, and A is a direct or indirect subclass of B.

(3) A is *not* a primitive type, and B is the type *Object*.

[40] This may be used to prevent cyclic inheritance when writing the typing rules for a type system.

(4) A is the *null* type, and B is not a primitive type.

(5) A and B are array types, and $A = A'$`[]` and $B = B'$`[]`, then $A' \prec_{\tau} B'$.

Rule (1) says that subtyping is a reflexive relation. Rule (2) and (3) are fairly obvious since every class in Espresso extends *Object*, therefore making subtying a transitive relation. Rule (4) says that *null* is a subtype of everything considering that we can assign the null reference to any reference type. We will return to rule (5) in Definition 8.

Now that we have everything in place, we can return to addressing the question regarding `a = b` and `b = a`. Because we declared a class A to be an extension of class B, we will have − in addition to all inheritance − that $A \prec_{\tau} B$. Any access to an instance of A, however, will be restricted to methods and fields declared in B, or, more broadly, in everything B inherits from or implements. Then, given that A is a subtype of B, the expression `b = a` is legal, but `a = b` is not, and since `b = a` is allowed, we say that Espresso arrays are **covariant** for reference types. This leads us to the next definition (following that of [5]).

Definition 8 (Covariant Arrays): If two references types α and β, $\alpha \prec_{\tau} \beta \Rightarrow Array(\beta, [\,]) :=_{\tau} Array(\alpha, [\,])$. For reference types, $\alpha \prec_{\tau} \beta$ is equivalent to $\beta :=_{\tau} \alpha$. □

That being the case, we must distinguish between references and other types, yielding the following equation for $T_v = Array(\beta, I_1)$ and $T_e = Array(\alpha, I_2)$

$$
\begin{aligned}
T_v :=_{\tau} T_e \Leftrightarrow \quad & \big(((\beta \sim_{\tau} \alpha) \wedge (\neg Class_?(\beta) \wedge \neg Class_?(\alpha))) \vee \\
& ((\alpha \prec_{\tau} \beta) \wedge (Class_?(\alpha) \wedge Class_?(\beta)))\big) \wedge \\
& ((I_1 = I_2) \vee ((I_1 = \bot) \wedge (I_2 = \bot))) \\
& (\text{If } e \text{ is an array literal, then } ALAC(T_v, e))
\end{aligned}
$$

Classes

As mentioned earlier, subtyping is a concept that is strongly linked to inheritance and polymorphism, and it provides a systematic approach to examine them. Before we leave this section, let us consider the following Espresso code:

```
B b = new A(...);
```

This assignment is legal if and only if $B = A$ or $A \prec_{\mathcal{T}} B$, making it easier to define assignment compatibility for reference types in Espresso; so for $T_v = Class(B)$ and $T_e = Class(A)$, we have

$$T_v :=_{\mathcal{T}} T_e \Leftrightarrow (B = A) \vee (A \prec_{\mathcal{T}} B)$$

Example 5: What if A not only extends B, but also implements several interfaces, as illustrate below?

```
interface I1 { }
interface I2 { }
class B { }
class A extends B implements I1, I2 { }
```

□

Well, the same rules apply to interface types as they do to classes when it comes to assignment compatibility. In this case, we have

$$A \prec_{\mathcal{T}} I_1 \text{ and } A \prec_{\mathcal{T}} I_2 \text{ an } A \prec_{\mathcal{T}} B$$

and there is no type relation existing between I_1 and I_2, between B and I_1, and between B and I_2 going $\uparrow$ (up) in the class hierarchy; however, there is a kind of lower bound relation going $\downarrow$ (down). In addition, it makes sense to define an upper and lower bound operator that imposes numerous inheritance restrictions, specially when dealing with parameterized class[41].

Definition 9 (Greatest Lower Bound): Similarly, we can also define a floor function for two reference types since $\prec_{\mathcal{T}}$ is defined for all valid combinations of reference types; therefore

$$\lfloor \alpha, \beta \rfloor_{\mathcal{T}} \stackrel{\text{def}}{=} \begin{cases} T & \text{if } \alpha \text{ and } \beta \text{ have anything in common} \\ \alpha & \text{if } \alpha \prec_{\mathcal{T}} \beta \\ \beta & \text{if } \beta \prec_{\mathcal{T}} \alpha \\ \bot & \text{otherwise} \end{cases}$$

[41] Generics are not available in Espresso, but they are in Java. *Jiapi* is designed to be used with any kind of DSL, so we have to briefly consider generic types.

the greatest lower bound GLB(α, β) is a lower bound T of α and β such that there is no other lower bound greater than T; T could also be one of the types α or β. In the same way as LUB, when constructing the greatest lower bound for more than two types, GLB($T_1, ..., T_n$) is defined as GLB(T_1, GLB($T_2, ..., T_n$)). Note that GLB is commutative, that is, GLB(α, β) = GLB(β, α). □

Consider the slightly modified version of the above Java code.

```
// same as before
class C extends A implements  I1, I2 { }
```

This results in the graph depicted in Figure 14. Note that we have turned it upside down to get the smallest class at the bottom, which is the class that the upper classes have in common.

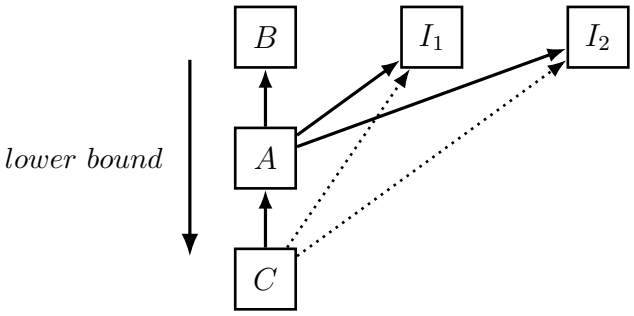


Figure 14. The associated class hierarchy in reverse.

It should be clear that the greatest lower bound (GLB) of two types is equivalent to the intersection of two types T_1 and T_2 (denoted as T_1 &$_\tau$ T_2). This means that intersection types are commutative and associative following the GLB properties. For example, T_1 &$_\tau$ $T_2 \Rightarrow T_2$ &$_\tau$ T_1, and T_1 &$_\tau$ (T_2 &$_\tau$ T_3) $\Rightarrow$ (T_1 &$_\tau$ T_2) &$_\tau$ T_3. Going back to Figure 14, the result of $\lfloor I_1, I_2 \rfloor_\tau$ (or I_1 &$_\tau$ I_2) is a type that is all of I_1 and I_2, enabling an object of this type, like C, to have all members of I_1 and I_2. As can be seen, an intersection is the result of "merging" several types into one. This allows us to combine many types in order to create a single type with all of the properties and features we need.

Naturally, the LUB operator (see Definition 6 on page 77) can be extended to work with

reference types (denoted as $\lceil \alpha, \beta \rceil_{\mathcal{T}}$); therefore, for two reference types α and β,

$$\lceil \alpha, \beta \rceil_{\mathcal{T}} \stackrel{\text{def}}{=} \begin{cases} T & \text{if } \alpha \text{ and } \beta \text{ have a common ancestor} \\ \alpha & \text{if } \alpha \prec_{\mathcal{T}} \beta \\ \beta & \text{if } \beta \prec_{\mathcal{T}} \alpha \\ \bot & \text{otherwise} \end{cases}$$

Likewise, a union type of two types (denoted $\alpha \mid_{\mathcal{T}} \beta$) is equivalent to the least upper bound; for example, the $\lceil B, C \rceil_{\mathcal{T}}$ (or $B \mid_{\mathcal{T}} C$) is B. Note that this operator can be used to assist in finding the most specific class in a multilevel inheritance hierarchy.

But how does the rule for subtypes and assignment work for generics? Does $\alpha \prec_{\mathcal{T}} \beta \Rightarrow G\langle\alpha\rangle \prec_{\mathcal{T}} G\langle\beta\rangle$? The answer is *no*; we will demonstrate this with a simple, yet reasonable, counter example.

Example 6: Consider the following Java-like code:

```
class α extends β { }
class G<E> { public E e; }
G<α> alpha = new G<α>();
G<β> beta = alpha;   ❶
beta.e = new β();   ❷
α a = alpha.e; // Error ❸
```

Why is there an error when $\alpha \prec_{\mathcal{T}} \beta$? Let us take it one step at a time, beginning from ❶. Presumably, since $\alpha \prec_{\mathcal{T}} \beta \Rightarrow G\langle\alpha\rangle \prec_{\mathcal{T}} G\langle\beta\rangle$, then ❶ should be legal. Considering that the type of `beta.e` is β (despite `beta` pointing to an object of type $G\langle\alpha\rangle$ from the preceding assignment), then ❷ should also be legal. So far, everything is going well with our assumptions, but what about the last assignment? The issue here is that we do not know what `alpha.e`'s true type should be in ❸; is it of type α or β? The compiler should not issue an error message if it is the former, but it should if it is the latter; so we say that $\alpha \prec_{\mathcal{T}} \beta \Rightarrow G\langle\alpha\rangle \prec_{\mathcal{T}} G\langle\beta\rangle$ defies the principles of substitution. Luckily, the Java compiler will generate a compile-time error from ❶.

$$\therefore \alpha \prec_{\mathcal{T}} \beta \not\Rightarrow G\langle\alpha\rangle \prec_{\mathcal{T}} G\langle\beta\rangle$$

□

Generics were introduced to Java in order to improve the expressiveness of its type system [224, 225, 226, 227]. However, Java generics include some characteristics that have been difficult to analyze and reason about in the past, such as Java wildcards [228, 229] (i.e, `<?>`), F-bounded generics[42] [226], and Java erasure [230]. For this reason, upper and lower bounds are frequently used to tackle some of these issues, limiting the types of values that can be passed to a class as **generic arguments**, as demonstrated next.

Remark 5: Java generics are implemented using *type erasure* [5, 225], which leads to all sorts of wacky issues, one of them being that we cannot find out what type a generic class is using at run-time[43]. To get around this problem and ensure generic type safety, we need to be able to specify unknown types as subtypes or super types. For example,

```
G<α> alpha = new G<α>();
G<β> beta = new G<β>();
G<? extends β> ebeta = alpha;  ❶
β b = ebeta.e; // legal  ❷
G<? super α> salpha = beta;  ❸
salpha.e = new α(); // legal  ❹
```

❷ is legal because if $\alpha \prec_{\mathcal{T}} \beta$ and `?` $\prec_{\mathcal{T}} \beta$, then `?` $\mapsto \alpha$ (i.e., `?` is an ancestor of the unknown type). Consequently, ❹ is also legal for the same reason, meaning, if $\alpha \prec_{\mathcal{T}} \beta$ and $\alpha \prec_{\mathcal{T}}$ `?`, then `?` $\mapsto \beta$ (i.e., `?` is the descendant of the unknown type). This implies that for any α such that $\alpha \prec_{\mathcal{T}} \beta$ (including $\beta = \alpha$), then:

- G<? **extends** β> can get as value G<α>
- G<? **super** α> can get as value G<β>

□

[42] F-bounded is an object-oriented technique that makes use of the type system to encode generic restrictions. Java, as opposed to Espresso, offers F-bounded subtyping, allowing type parameters to generic classes to have bounds that specifically refer to the parameter; for example, `class A<T extends B<T>> { ... }`

[43] For example, at a later stage in the compilation process, G<E> is replaced by G<*Object*>; thus, at run-time, the notion of E is lost – it is **only** *Object* by then. This is known as the "erasure" system, in which E is used to check the sources early in the compilation process before being replaced by *Object*.

Recall from Definition 3 on page 69 that we can obtain any specified type by assigning a type to the type variable α, and potentially a range as well. Now, how do we deal with *generic* or *parameterized classes* like the one below:

```
public class Hastable<A, B> {
  ...
}
```

Obviously, the type of the expression that creates a `Hashtable` is $Class(...)$; but, what should `...` be? It is impossible to instantiate a `Hashtable` without the types `A` and `B` being specified. For example, if we had the expression

```
new Hashtable<Integer, String>
```

The types *Integer* and *String* must be included in the type for the expression. This is simple to accomplish if we use type variables to give the type of the `Hashtable` class as seen below:

$$Class(Hashtable\texttt{<}\alpha, \beta\texttt{>})$$

where α represents the type `A` and β represents the type `B`. Therefore, when determining the type of `new Hashtable<Integer, String>`, we substitute α and β for the appropriate types, yielding:

$$Class(Hashtable\texttt{<}Integer, String\texttt{>})$$

A number of terms that are important to understand when working with types and type constructors are defined in Definition 10 [5].

Definition 10 (Reifiable and Variance): Any type whose type information is entirely available at runtime is referred to as a **reifiable** type. This covers non-generic types and primitives; nevertheless, any type that lacks complete runtime access to its information is said to be **non-reifiable**. Furthermore, **variance** in a type system refers to the relationship between subtyping between complex types and subtyping between the parts that make up the complex types. For example, the type constructor C is said to be **covariant** if there

is a type constructor C, two types A and B where $A \leq_{\mathcal{T}} B$, and the condition that if $A \leq_{\mathcal{T}} B \Rightarrow C(A) \leq_{\mathcal{T}} C(B)$. If, however, $A \leq_{\mathcal{T}} B \Rightarrow C(A) \geq_{\mathcal{T}} C(B)$, then we say that C is **contravariant**. Note that it is referred to as **bivariant** if both applied, but if neither applies, then it is referred to as **invariant** or **nonvariant**. □

We will stop here as we have covered everything we will need in order to write parts of the type system specification of Espresso − and latter of Espresso using *Jiapi*. Table 5 summarizes the rules for assignment compatibility that we have just defined [5].

Table 5. Assignment compatibility ($:=_{\mathcal{T}}$).

Type	Assignment Compatible
record	$T_v \sim_{\mathcal{T}} T_e$
union	$T_v \sim_{\mathcal{T}} T_e$
array	$\big(((\beta \sim_{\mathcal{T}} \alpha) \wedge (\neg Class_?(\beta) \wedge \neg Class_?(\alpha))) \vee$ $((\alpha \prec_{\mathcal{T}} \beta) \wedge (Class_?(\alpha) \wedge Class_?(\beta)))\big) \wedge$ $((I_1 = I_2) \vee ((I_1 = \perp) \wedge (I_2 = \perp)))$ (If e is an array literal, then $ALAC(T_v, e)$)
enum	$T_v \sim_{\mathcal{T}} T_e$
procedure	$(m_1 = m_2) \wedge (\bigwedge_{i=1}^{m_1} t_{1,i} \sim_{\mathcal{T}} t_{2,i}) \wedge (t_1 \sim_{\mathcal{T}} t_2)$
pointer	$T_v \sim_{\mathcal{T}} T_e$
class	$(T_v = T_e) \vee (T_e \prec_{\mathcal{T}} T_v)$
named type	$T_v \sim_{\mathcal{T}} T_e$

4.4.6 Type System Specification

The Espresso type system is described in terms of an abstract syntax that omits a lot of the details found in the concrete syntax. The abstract syntax is somewhat analogous to the parse-tree data structures used by the compiler, although it excludes a number of derived forms that can be expressed in terms of more basic syntax. Let us start with the Espresso$^{\pi}$ type checking specification. Recall, the typing judgment for the terms has the form:

$$\mathbf{\Lambda}, C \vdash E : T \quad \text{Term } E \text{ has type } T \text{ in context } C \text{ and in a type environment } \mathbf{\Lambda}$$

We need a method to check whether a given Espresso program in the typed λ-calculus is well-typed, or whether it is guaranteed to behave properly. Being well-typed is a judgment, much like we have for first-order logic. Here our judgments are of the form $\mathbf{\Lambda}, C \vdash E : T$, meaning, "expression E has type T under the set of hypotheses $\mathbf{\Lambda}, C$". Assumptions about the many types of variables, such as $x_1 : T_1, x_2 : T_2, ..., x_n : T_n$, etc., make up the hypotheses that guide our judgments. Here, $x_1, ..., x_n$ are *distinct* variables (or identifiers) and $T_1, ..., T_n$ are types. A list of this form is called *typing context*. Such as list is supposed to serve as the "context" for type checking terms. In order to judge the expression in the typed λ-calculus, we created a set of inference rules. These rules are standard for methods, fields, classes, statements, and expressions.

As in Java, static type checking is performed at compile-time in Espresso, thus these rules are invoked as subroutines during type checking (typically in the form of a *visit* [3, 5]). In addition, the methods in Table 6 provide information about the relative position of a variable within the current scope during type checking. We assume that the abstract syntax complies with a number of syntactic constraints that are discussed in the next section. Note that a significant component of this project is to develop the skills needed to produce an implementation from a specification. For this reason, the following *specification* is a description of a property (*yes/no* question; *true/false* statement). It does **not** specify how to determine whether certain properties holds, though it may make suggestions in this regard.

Table 6. Helper methods

Method	Description
$classname(\textbf{class}\ cn\ \texttt{\{ ... \}}) = cn$	Returns the name of a class declaration
$hierarchy(\textbf{class}\ cn\ \texttt{\{ ... \}}) = \{cn, cn'\}$ $hierarchy(\textbf{class}\ cn\ \texttt{\{ ... \}}) = \{\emptyset\}$	Returns a set containing the name a class and its parent class
$method\_name(T\ mn\ \texttt{(}\overline{T\ x}\texttt{) \{ ... \}}) = mn$	Returns the name of a method declaration
$field\_name(T\ fn) = fn$	Returns the name of a field declaration
$\frac{\forall i \in 1..n : \forall j \in 1..n : id_i = id_j \Rightarrow i = j}{unique\_id(id_1, ..., id_n)}$	Determines if the identifiers in a list are pairwise disjoint

Table 6. Helper methods

$\dfrac{\neg(\exists j_1, ..., j_k \in 1..n : (\forall i \in 1..k-1 : id_{j_{i+1}} = id'_{j_{i+1}}) \vee (id_{j_k} = id'_{j_1}))}{acyclic(\{(id_1, id'_1), ..., (id_n, id'_n)\})}$	Determines if a set of pairs contain no cycles
$\dfrac{\textbf{class } cn \texttt{ \{ ... \} } \in P}{fields(cn) = \{\!\{id_1 : T_1, ..., id_n : T_n\}\!\}}$	Defines a type environment constructed from the fields of K
$\dfrac{\textbf{class } cn \texttt{ \{ ... } M_1 \text{ ... } M_n \texttt{ \} } \in P \quad \exists j \in 1..m : method\_name(M_j) = mn \quad mn = T\ mn(T_1\ x_i ... T_n\ x_n) \texttt{ \{ ... \} }}{find\_method(K, mn) = (T_1, ..., T_n) \to T}$ $\vee$ $\dfrac{\textbf{class } cn \texttt{ \{ ... } M_1 \text{ ... } M_n \texttt{ \} } \in P \quad \forall j \in 1..m : method\_name(M_j) \neq mn}{find\_method(K, mn) = \bot}$	Determines if a method declaration of name mn exists in K, or $\bot$ if no such method exists
$\dfrac{\textbf{class } cn \texttt{ \{ ... } F_1 \text{ ... } F_n \texttt{ \} } \in P \quad \exists j \in 1..m : field\_name(F_j) = fn \quad fn = T\ fn}{find\_field(K, fn) = T\ fn}$ $\vee$ $\dfrac{\textbf{class } cn \texttt{ \{ ... } F_1 \text{ ... } F_n \texttt{ \} } \in P \quad \forall j \in 1..m : field\_name(F_j) \neq fn}{find\_field(K, mn) = \bot}$	Determines if a field declaration of name fn exists in K, or $\bot$ if no such field exists
$\dfrac{find\_method(K, mn) \neq \bot \Rightarrow find\_method(K, mn) = find\_method(K', mn)}{overloading(K, K', mn)}$	Determines if a method is overloaded

4.4.6.1 Notation for Rules

We use the flowing notation

$$\frac{hypothesis_1 \quad hypothesis_2 \quad ... \quad hypothesis_n}{conclusion}$$

where in a rule there is only ever one possible conclusion; nevertheless, the number of premises is limitless. For example, the above is a rule that states that if we can derive all of $hypothesis_1, hypothesis_2, ..., hypothesis_n$, then we can also conclude the *conclusion*. It should be said that a special case arises when $n = 0$, that is, we write this case as

$$\frac{}{conclusion}$$

or we can omit the horizontal bar and write just

$$conclusion$$

representing an *axiom*. The process of starting with one or more axioms, possibly applying certain rules, and ending with a conclusion is known as a *derivation*. A typing derivation is a tree structure of inferences that each aim to demonstrate how a certain type of rule applies to the term in question, as depicted in Example 7.

Example 7: Figure 15 gives a typing derivation for the *while* statement rule mentioned in Section 4.4.6 on page 102 (using parts of Figure 4) where the root of the tree is the whole expression, each node is an instance of a typing rule, and leaves are the rules with no hypotheses.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{\Lambda}_{\mathcal{T}}(x) = \texttt{boolean}}}{\mathbf{\Lambda},\, C \vdash x : \texttt{boolean}}}{\mathbf{\Lambda},\, C \vdash \mathbf{!}x : \texttt{boolean}} \qquad \dfrac{\dfrac{}{\mathbf{\Lambda}, C \vdash 7 : \texttt{int}} \qquad \dfrac{\dfrac{}{\mathbf{\Lambda}, C \vdash 3 : \texttt{int}} \quad \dfrac{}{\mathbf{\Lambda}, C \vdash 2 : \texttt{int}}}{\mathbf{\Lambda}, C \vdash (3 + 2) : \texttt{int}}}{\mathbf{\Lambda}, C \vdash 7 + (3 + 2) : \texttt{int}}}{\mathbf{\Lambda},\, C \vdash \textbf{while (} \mathbf{!}x \textbf{)}\ 7 + (3 + 2)}$$

Figure 15. Proof tree for `while` statement.

Although the suitable formalism for type checking is logical rules of inference, Figure 16 offers another perspective on the aforementioned derivation.

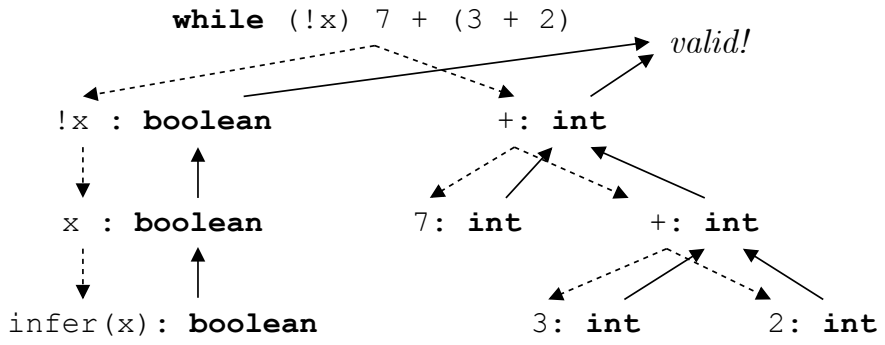


Figure 16. The typing reason as a tree.

Another option is to write the derivation in a linear form by labeling the statements as we write them and making references to them from other statements as needed. However, in actual use, it is rarely required to explicitly write out typing derivations in full. Annotating terms and subterms is sufficient, but we must make sure that the annotation adheres to the type requirements. □

Example 7 demonstrates a trace of actions the type checker performs, built up rule-by-rule, with each judgment being a conclusion from the judgments that came before it (with some of the rules indicated beside the line). Consequently, a **derivation** (or **proof tree**) is a tree where nodes are instantiations of inference rules and edges connect a premise to a conclusion. The goal of the type checker is to verify that such derivation exists. This derivation in turn provides a specification for the static correctness of Espresso programs. Bear in mind that we only take into account instances of judgments that are well-formed; if an instance is not well-formed, it has no meaning and it is pointless to evaluate whether it holds or whether a derivation of it exists.

4.4.6.2 Syntactic Restrictions

The type checker imposes a number of syntactic limitations. Some of these properties could have been provided in the grammar, but doing so would have rendered it significantly verbose. Others are properties that could be specified as part of the typing rules, but it is easier to specify them separately. Let us have a quick look at them.

1. **Context:** Any of the Espresso types are permissible for variables, which are then gathered into a context (this being a symbol table)

2. **Expressions:** Checking expressions is defined in terms of type inference, so for

$$x : \texttt{int}, y : \texttt{int} \vdash x * y > 10 : \texttt{boolean}$$

   $x * y > 10$ is a *boolean* expression in the context where x and y are *integer* variables. From this, it is clear that an inference rule specifies a unique way for determining whether a given expression has a particular type. For example, the $\Gamma \vdash n : \texttt{int}$ rule of

n : `int` is *axiomatic* − a numeric constant has type *int* under all conditions; therefore, 10 is a constant *integer* value. Evidently, it is also clear that from the binary rule for multiplication that if two subexpressions are integers, then the binary operation on those subexpressions is also an integer, allowing type checking of the entire expression.

3. **Expression statements:** We only need to know that an expression has a type in order to infer its type; this often applies to assignments and method calls.

4. **Method definitions:** The variables listed as the method's parameters define the context, and therefore the statements $\overline{S}^{i \in \{1..n\}}$ in the method body are checked in this context. It should be noted that the type checker verifies that each variable in the parameter list is unique and that declarations may cause the context in the body to change.

5. **Statements:** When type checking a statement, we are only concerned with determining whether the statement is valid and not with its type. For instance, validating a while statement entails type checking as well as validating its constituent expressions.

    - *Return statement:* We can inspect the method body for a return statement of the expected type when checking a method definition. A more sophisticated version of this could also allow returns in ternary expressions.
    - *Declarations and blocks structures* Every declaration has a scope that is within a certain block. Since portions of code between curly brackets, { and }, are roughly equivalent to blocks in Espresso, two rules govern the use of variables:
        (a) A variable declared in a block has its scope until the end of that block.
        (b) A variable can be declared again in a inner block, but not otherwise.

    Whenever a new variable is declared, it is first verified that it is unique in the context before being added to the current scope. As a result, a declaration extends the context in which the statements that follow are checked. Strictly speaking, if the statements that come after a declaration are valid in the context in which the declared variable

is added, then the declaration is valid. This addition causes the type checker to recognize the effect of the declaration. Naturally, the declaration only has an impact on the statements that are already part of the same context (i.e., a block); subsequent statements are unaffected.

6. **Overloading:** Binary arithmetic operations (such as +, −, ⋆ and /) and comparisons (such as ==, !=, <, >, <= and >=) are in many languages **overloaded** which means that they are usable for different types. For example, Figure 17 gives the typing rules for Expresso if the possible types are `int`, `double`, or `string`.

$$\frac{\mathbf{\Lambda}, C \vdash E_1 : T \qquad \mathbf{\Lambda}, C \vdash E_2 : T}{\mathbf{\Lambda}, C \vdash E_1 + E_2 : T} \textbf{if } T \in \{\texttt{int, double, string}\}$$

$$\frac{\mathbf{\Lambda}, C \vdash E_1 : T \qquad \mathbf{\Lambda}, C \vdash E_2 : T}{\mathbf{\Lambda}, C \vdash E_1 = E_2 : \texttt{boolean}} \textbf{if } T \in \{\texttt{int, double}\}$$

Figure 17. Possible typing rules for `int`, `double`, or `string`.

It should be noted that since strings in Espresso are immutable, the == operator for string comparison does not compare the contents of the string (it only compares memory address). For this reason, the second typing rule does not apply to strings.

4.4.6.3 Type Checking a Program

At the top level of the program, we need to figure out the types of classes and make sure their bodies are well-typed. Therefore, we define Espresso programs as being well-formed if they go through three phases of type checking:

1. Generate a class environment
2. Check that the class environment is well-formed, and
3. Check that every class is well-formed under the environment

1. **Phase 1:** Generating a class environment is straightforward. For each class definition, we make a binding that associates the class name *cn* with an entry, for example: `class`(K, C). This entry is made of of the class and a local environment C that records the type of the super class and the types that each class member has (fields and methods).

2. **Phase 2:** A class environment is well-formed if it meets the following criteria:

   (a) A top element (often *Object*) is present in the partial order of ground types defined by $\prec_{\mathcal{T}}$.

   (b) For two ground types, if K_1 is well-formed and $\vdash K_1 \prec_{\mathcal{T}} K_2$, then K_2 is well-formed.

   (c) The type is the same for method declarations with the same name across classes. (This is merely done for the sake of illustration.)

3. **Phase 3:** We presume that we have a well-formed class environment. We start with the rule for typing expression, followed by the rules for typing statements. We then give rules for all remaining Espresso constructs.

Recall, we carry around an environment (some kind of symbol table) that records the type of the variable symbols (variables and class names) during type checking. The initial environment contains the type of all classes defined so far (including those of the Espresso library). Let us start at the top-level, type checking a program. A program P type checks (written $\vdash P$) if all the classes in that program type check. (Recall that a program is simply a collection of classes.) A class type K checks (written $\vdash K$) if all of fields and methods it contains type check as well (i.e. $C \vdash F$ and $C \vdash M$). To type check a program, we type check each class in the program, ensuring that all classes in the program are added to the environment. Although there is a circularity here, it turns out to be not a problem.

To type check a class, we must type check all of its methods and fields. Let us focus on the methods for now. A method type checks (written $C \vdash M$ or $C \vdash T\ mn\ (\overline{T\ x})$ `{` S **return** E `;` `}`) if all the statements and declarations it contains type check. When type checking, for example, *System.out.println(...)*, the environment contains the name of the

class where this invocation was made (with a type indicating it is a class with a method of type `void`), as well as the name *System*, which is a class with a field *out* pointing to an object and a method *println* expecting a *String*.

Let us assume that *println* is invoked with *a.toString()*. Then, *a* must either have a type *Object* or some type $T \prec_{\mathcal{T}} Object$ because both have a method called *toString*. Thus, *a.String()* is valid, and the result is of course a string, which means that the call to *println* is also valid and returns no value (i.e., `void`). Therefore, this statement type checks and can be represented as depicted in Figure 18.

$$\frac{\begin{array}{l}\mathbf{\Lambda}, C \vdash \texttt{System.out} : \texttt{PrintStream \{ ... void println(...) ... \}} \\ \mathbf{\Lambda}, C \vdash \texttt{a.toString()} : \texttt{String}\end{array}}{\mathbf{\Lambda}, C \vdash \texttt{System.out.println(a.toString())} : \texttt{void}}$$

Figure 18. Type checking a Java program.

So what does Figure 18 says, in other words, what does type checking guarantees? It says that certain bad things cannot happen at runtime. It is worth noting that every object at runtime has a corresponding runtime class K (the class it was created as). $\vdash P$ is guaranteed, that is, whenever the type system attempt to invoke some method M on some object O during program execution, object O has an implementation of method M; otherwise, it is an error.

4.4.6.4 Subtyping

The subtyping relation on class names is the least partial order consistent with the rules given in Figure 19. We use $\prec_{\mathcal{T}}$ to denote the subtyping relation and say that T_1 is a subtype of T_2 if $T_1 \prec_{\mathcal{T}} T_2$. Note that $\prec_{\mathcal{T}}$ is reflexive (1) and transitive (2), and generated by the **extends** relation among classes (3), as defined in Definition 7.

$$\frac{}{T \prec_{\mathcal{T}} T}(1) \qquad \frac{T_1 \prec_{\mathcal{T}} T_2 \quad T_2 \prec_{\mathcal{T}} T_3}{T_1 \prec_{\mathcal{T}} T_3}(2) \qquad \frac{\textbf{class } K_0 \textbf{ extends } K_1 \texttt{ \{ ... \}}}{K_0 \prec_{\mathcal{T}} K_1}(3)$$

Figure 19. Type checking subtypes.

How do we implement the $\prec_T$ operator that we have described? There are many situations to consider, but Table 7 summarizes several of them according to the hierarchy for types given in Figure 20.

Table 7. Implementing $\prec_T$ with different types

From \ To	Class Type	Primitive Type	Array Type	Null Type	Error Type
Class Type	If same or inherits from	No	No	No	No
Primitive Type	No	No	No	No	No
Array Type	No	No	If underlying types match	No	No
Null Type	Yes	No	No	Yes	No
Error Type	Yes	Yes	Yes	Yes	Yes

One challenging problem is what to do when an expression does not have a valid type under the rules. A simple recovery mechanism is to assign the type **Error** to any expression that cannot otherwise be given a type (we used this method in Espresso). Although not previously discussed, we introduce a new type representing an error into the type system. The **error type** is less than all other types and is denoted $\perp$ (sometimes referred to as **bottom type**). By definition, $\perp \prec_T T$ for any type T, so on discovery of a type error we assume that the expression − one that produces an error − has type **error type**.

If the intended outcome of the type operation was to produce a type itself, any type operations where one or more of the operands is an error type will and should produce both an error message and an error type.

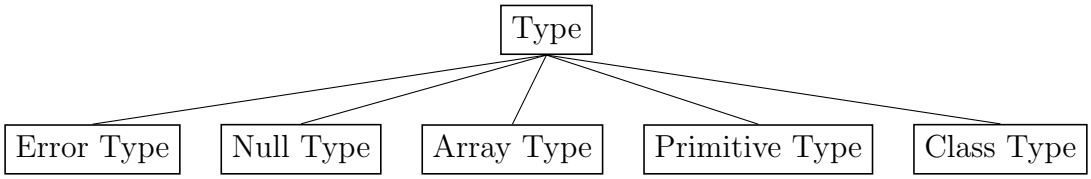


Figure 20. Hierarchy of types in Espresso.

4.4.6.5 Type Checking Expression

We must first define how we choose an expression's type in this conceptual framework for type systems. The context C in the type environment $\mathbf{\Lambda}$, which maps names to types, serves as a representation of what bound variables and methods are in scope for type checking expressions. We use the metavariable symbol *id* to represent arbitrary identifiers, n to represent an integer literal constant, *string* to represent a string literal constant (recall that a string is an *Object* in Java), *char* to represent a character literal constant, and *true* or *false* to represent boolean literal constants. Using these conventions, the expression typing rules are depicted in Figure 21.

4.4.6.6 Type Checking Statements

To typecheck statements, we need all the information used to typecheck expressions. Statements can also result in new variable bindings because they contain declarations, which results in an updated type context that we will refer to as C' (although most of the statements are fairly straightforward and do not change C). To represent a statement, we utilize the same metavariable S and use the statement typing rules given in Figure 22.

4.5 Closing Remarks

Espresso's type system at a basic level works in the same way as typed λ-calculus. During assignments, programmers provide explicit type annotations to variables, and if the type of the expression is anything other than the type, Espresso complains and stops compilation. Using an operator on an incorrectly typed object, calling a function on incorrectly typed inputs, or defining a function to return a type other than its explicitly declared return type are all catchable compilation errors (hence the statically typed nature of Espresso). A second, maybe less evident, point is that because the Espresso type system is a first-order type system, it does not support the passing of methods as parameters or the return of methods as results, but it is feasible to pass and return objects that include methods.

Object identity is another intriguing aspect of the language. In Espresso, this is useful, but is it rarely what we want? For example, we may want to say that two lists are equal if they

have the same sequence of elements (This is a little bit like in set theory, where two sets are considered equal if they have the same elements, or considering two lists equals if they have the same elements in the same order.) To have object equality in the language, we need the object's equals method to satisfy the main properties of equality: **reflexivity**, **symmetry**, and **transitivity**. Naturally, even with a more robust type system (or simply the presence of a static type system in comparison to dynamically typed languages), languages like Espresso cannot capture all errors during compilation.

$$\frac{}{\mathbf{\Lambda},\, C \vdash n : \texttt{int}} \qquad \frac{}{\mathbf{\Lambda},\, C \vdash \mathit{string} : \texttt{String}} \qquad \frac{}{\mathbf{\Lambda},\, C \vdash \mathbf{true} : \texttt{boolean}}$$

$$\frac{}{\mathbf{\Lambda},\, C \vdash \mathbf{false} : \texttt{boolean}} \qquad \frac{}{\mathbf{\Lambda},\, C \vdash \mathit{char} : \texttt{char}}$$

$$\frac{C \neq \bot}{\mathbf{\Lambda},\, C \vdash \mathbf{this} : C} \qquad \frac{}{\mathbf{\Lambda},\, C \vdash \mathbf{new}\ cn\ \texttt{()}\colon cn} \qquad \frac{id \in dom(\mathbf{\Lambda})}{\mathbf{\Lambda}, C \vdash id : \mathbf{\Lambda}(id)}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E_1 : \texttt{int} \qquad \mathbf{\Lambda},\, C \vdash E_2 : \texttt{int} \qquad \odot \in \{+, -, \times, \div\}}{\mathbf{\Lambda},\, C \vdash E_1 \odot E_2 : \texttt{int}}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E_1 : \texttt{int} \qquad \mathbf{\Lambda},\, C \vdash E_2 : \texttt{int} \qquad \odot \in \{=, \neq, <, \leq, >, \geq\}}{\mathbf{\Lambda},\, C \vdash E_1 \odot E_2 : \texttt{boolean}}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E_1 : \texttt{boolean} \qquad \mathbf{\Lambda},\, C \vdash E_2 : \texttt{boolean} \qquad \odot \in \{=, \neq, <, \leq, >, \geq\}}{\mathbf{\Lambda},\, C \vdash E_1 \odot E_2 : \texttt{boolean}}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : \texttt{int}}{\mathbf{\Lambda},\, C \vdash -E : \texttt{int}} \qquad \frac{\mathbf{\Lambda},\, C \vdash E : \texttt{boolean}}{\mathbf{\Lambda},\, C \vdash !E : \texttt{boolean}}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E_1 : T\texttt{[]} \qquad \mathbf{\Lambda},\, C \vdash E_2 : \texttt{int}}{\mathbf{\Lambda},\, C \vdash E_1\texttt{[}E_2\texttt{]} : T} \qquad \frac{\mathbf{\Lambda},\, C \vdash E : \texttt{int}}{\mathbf{\Lambda},\, C \vdash \mathbf{new}\ T\texttt{[}E\texttt{]} : T\texttt{[]}}$$

$$\frac{\begin{array}{l}\mathbf{\Lambda},\, C \vdash E' : K \qquad \mathit{find\_method}(K, id) = (T'_1, ..., T'_n) \to T \\ \mathbf{\Lambda},\, C \vdash \overline{E_i : T_i}^{\,i \in \{1..n\}} \qquad \overline{T_i}^{\,i \in \{1..n\}} \leq_{\tau} \overline{T'_i}^{\,i \in \{1..n\}}\end{array}}{\mathbf{\Lambda},\, C \vdash E'.id\texttt{(}\, E_1, .., E_n\,\texttt{)} : T}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : K \qquad \mathit{find\_field}(K, id) = id : T}{\mathbf{\Lambda},\, C \vdash E.id : T}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : T\texttt{[]}}{\mathbf{\Lambda},\, C \vdash E\texttt{.length} : \texttt{int}}$$

Figure 21. Type checking expressions.

$$\frac{\mathbf{\Lambda},\, C \vdash \overline{S_i}^{\,i \in \{1..n\}}}{\mathbf{\Lambda}, C \vdash \texttt{\{}\, S_1, ..., S_n \,\texttt{\}}} \qquad \frac{\mathbf{\Lambda}(id) = T_1 \qquad \mathbf{\Lambda},\, C \vdash E : T_2 \qquad T_2 \prec_{\boldsymbol{\tau}} T_1}{\mathbf{\Lambda},\, C \vdash id = E\ \textbf{;}}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : \texttt{boolean} \qquad \mathbf{\Lambda},\, C \vdash S_1,\, C' \qquad \mathbf{\Lambda},\, C \vdash S_2,\, C''}{\mathbf{\Lambda}, C \vdash \textbf{if}\ \texttt{(}\, E \,\texttt{)}\ S_1\ \textbf{else}\ S_2 : C' \sqcup C''}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : \texttt{boolean} \qquad \mathbf{\Lambda},\, C \vdash S,\, C'}{\mathbf{\Lambda}, C \vdash \textbf{if}\ \texttt{(}\, E \,\texttt{)}\ S : C'}$$

$$\frac{\mathbf{\Lambda},\, C \vdash E : \texttt{boolean} \qquad \mathbf{\Lambda},\, C \vdash S,\, C'}{\mathbf{\Lambda}, C \vdash \textbf{while}\ \texttt{(}\, E \,\texttt{)}\ S} \qquad \frac{\mathbf{\Lambda},\, C \vdash E : T}{\mathbf{\Lambda}, C \vdash \textbf{return}\ E\ \textbf{;} : T}$$

Figure 22. Type checking statements.

Chapter 5

Jiapi

5.1 Introduction

As briefly mentioned in Section 1.4 on page 13, *Jiapi* is a type checker generator that uses first-order logic and set notation, along with an expression grammar for describing a type system. To define a type system, *Jiapi* mixes a controlled form of a meta-language with the concepts of *atomic* (or primitive) and *constructed* (or structured) types. The idea is to make the automation of code generation for semantic checking tasks possible for which types alone can be specified and inferred. This approach connects objects (instances of classes) with various components in the type system through an iterative process. The iterative nature of this process can be explained by the fact that an object representing a **semantic node** (or parse tree node) is often implemented by a number of sub-objects, much like a procedure is typically implemented by a number of procedures in top-down programming [231, Section 7][44].

In this chapter we briefly explore a few standard constructs of mainstream languages as presented by [5], and for which we can specify the required checks in the context of **type predicates** (of the form $\mathcal{P}_?(x)$), **type equality** (the $=_\tau$ operator), **type equivalence** (the $\sim_\tau$ operator), and **assignment compatibility** (the $:=_\tau$ operator). Other operators that are either derivable or extensions of the ones already stated are also available. Furthermore, fragments of the meta-language's syntax are specified in Extended Backus-Naur Form

[44] The difference here is that, in the design process, functionality and data representation can be both improved.

(EBNF) as needed. We start by outlining the notation that the tool uses. The process for generating the `*.java` files is then described. Next, the structure of a *Jiapi* file is explained in detail, step-by-step. Finally, we conclude the chapter with a process-oriented language constructed type example and a simple representation of typed λ-calculus rules to *Jiapi*.

5.2 Notation

We encounter the challenge of varied notation when referring content from various fields of type literature. Clearly, there are two alternatives to consider: Either cite all sources using the same notation or transcribe the sources to a common notation. In this thesis, we decided to go with the latter alternative for three reasons. First, the overall structure and format will be more consistent. Second, we want to stress commonness while avoiding being distracted by minor differences in notation. Third, the changes required are minor, primarily consisting of symbol replacement.

Table 8 emphasizes the naming convention and notation that will be followed. This notation corresponds to the *syntax* for specifying a type system in *Jiapi* with a few minor exceptions for better readability and clarity. We include the original content when the structure of the supplied material needs to be altered in non-trivial ways to illustrate a commonality.

Table 8. Naming convention and notation.

Symbol	
T	Type with unknown nullability
$\{T\}$	Set of all possible values of T
{: ... :}	Code declaration section
{...}?	Semantic predicate
(...)	Sequence of elements
{...}	Optional sequence of elements
[...]	Required sequence of elements
?	Optional closure
+	Positive closure
*	Kleen closure
..	Range operator
$\langle ... \rangle$	Sequence formation
$s[k]$	k^{th} member of the sequence s

Table 8. Naming convention and notation.

$s - k$	Drop the first k member of the sequence s
$s + t$	Concatenation of sequences s and t
$t \Rightarrow a, b$	McCarthy-like conditional "if t then a else b"
Well-formed Type	
Γ	Type environment
$A =_{\mathcal{T}} B$	A and B have the same type
$A \sim_{\mathcal{T}} B$	A value of type A can be assigned to a variable of type B and a value of type B can be assigned to a variable of type A
$A :=_{\mathcal{T}} B$	A value of type B can be assigned to a variable of type A but a value of type A cannot be assigned to a variable of type B
$A <_{\mathcal{T}} B$	A precedes B
$A \leq_{\mathcal{T}} B$	A value of type A can legally be assigned to a variable of type B
$A \prec_{\mathcal{T}} B$	A is a subtype of B
$A \not\prec_{\mathcal{T}} B$	A and B are not related with respect to subtyping
$\lfloor A, B \rfloor_{\mathcal{T}}$	Greater lower bound (or GLB) of A and B
$\lceil A, B \rceil_{\mathcal{T}}$	Least upper bound (or LUB) of A and B
Builder Notation	
$\bar{a} = a_1, ..., a_n$	Sequence of elements
$A^{s+1} = A^s \cup \{a_i\}$	Adding an elements to A
$A^{s-1} = A^s \backslash \{a_i\}$	Removing an element from A
$\lvert A \rvert$	Cardinality (or size) of A
$!A$	Complement of A are all elements which are not in A
$!(A \wedge B)$	Complement of A and B are all elements which are not in A and B
$A \backslash B$	A set-minus B are all elements in A which are not in B
$A \subset B$	A is a proper subset of B
$A \not\subset B$	A is not a proper subset of B
$A \subseteq B$	A is contained in B
$A \not\subseteq B$	A is not contained in B
$A \cap B$	A intersection B are all the members of A that also belong to B
$A \cup B$	A union B are all the members of A and all the members of B
$A \leftarrow B$	B is assigned to A

Table 8. Naming convention and notation.

$a \in A$	The element a is a member of the set A
$a \notin A$	The element a is not a member of the set A
$\mathbb{A} = \{a \in A \mid \{...\}?\}$	Removes unwanted elements from A based on a *semantic predicate* and returns a new set $\mathbb{A}$
Textual Symbol	
$=_T$	`=`T
$\sim_T$	$\sim T$
$:=_T$	`:=`T
$<_T$	`<`T
$\leq_T$	`<=`T
$\prec_T$	`<:`T
$\not\prec_T$	`<:>`T
$\lfloor \_, \_ \rfloor_T$	GLB(_,_)
$\lceil \_, \_ \rceil_T$	LUB(_,_)
$\{a_1, ..., a_n\}$	{ , } to enclose elements of a set
$\emptyset$	{ } (empty set)
$\forall$	*for all, for every, for any*
$\in$	*in*
$\notin$	`!` *in*
$\subset$	*subset*
$\not\subset$	`!` *subset*
$\bigcup$	*union*
$\bigcap$	*intersection*
$\bigwedge$	*and*
$\bigvee$	*or*
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
Others	
`//`	Single-line comment
`/*...*/`	Multi-line comment

[1] *low* is the lowest legal index, and *high* is the highest index.
[2] This notation is borrowed from [232].

Predicates have a number of algebraic properties that can be useful when examining and transforming logic formulations. Predicate symbols, such as p and q, represent object-to-object relations. Keep in mind that a relation is a collection of tuples. For example, we could define the relation {(wine, red),(ocean,blue),...}. However, without an interpretation, we have no idea what the relation or objects are. Table 9 highlights the various types of expressions that can be employed in propositional and predicate logic, as well as their

meanings.

Table 9. Logic notation.

Logic notation	
true, *false*	*Boolean* (truth) constants
p, q, ...	*Boolean* variables
$\mathcal{P}_?(x)$, $\mathcal{Q}_?(x)$, ...	*Boolean* predicate
$\neg p$	Negation of p
$p \wedge q$	Conjunction of p and q
$p \vee q$	Disjunction of p and q
$\mathcal{P}_?(x) \wedge \mathcal{Q}_?(x)$	Conjunction of predicate $\mathcal{P}$ and predicate $\mathcal{Q}$
$\mathcal{P}_?(x) \vee \mathcal{Q}_?(x)$	Disjunction of predicate $\mathcal{P}$ and predicate $\mathcal{Q}$
$\mathcal{P}_?(x) \Rightarrow \mathcal{Q}_?(x)$	Implication: predicate $\mathcal{P}$ implies predicate $\mathcal{Q}$
$\mathcal{P}_?(x) \Leftrightarrow \mathcal{Q}_?(x)$	Logic equivalence $\mathcal{P}$ implies predicate $\mathcal{Q}$
Set-builder notation	
$a \in A$	Set membership. The element a is a member of the set A
$a \notin A$	Set membership. The element a is *not* a member of the set A
$\{x : \mathcal{P}_?(x)\}$	Set of all elements x that makes the predicate $\mathcal{P}$ true
$A = B \stackrel{\text{def}}{=} x \in A \Leftrightarrow x \in B$	Set equivalence. Any member of set A is also a member of set B, and any member of set B is also a member of set A
$A \subseteq B \stackrel{\text{def}}{=} x \in A \Rightarrow x \in B$	Any member of set A also is a member of the set B. However, there may be members of B that are *not* in A, although A and B can be the same.
$A \subset B \stackrel{\text{def}}{=} A \subseteq B \wedge A \neq B$	Any member of A also is a member of B. However, there exists at least one member of B that is *not* a member of A
$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \vee x \in B\}$	Any number of A or B or both is a member of the union of A and B
$A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \in B\}$	Any member of A that also is a member of B is in the intersection of A and B.
$A \backslash B \stackrel{\text{def}}{=} \{x \mid A \wedge x \notin B\}$	Any member of A that is *not* a member of B is a in the difference of A and B
$\forall x \ \mathcal{P}_?(x)$	Universally quantified predicate expression

Table 9. Logic notation.

$\exists x\ \mathcal{P}_?(x)$	Existentially quantified predicate expression

5.3 Meta-Language

In order for *Jiapi* to produce a type checker for a language, its type system must be described by a meta-language. This means that we need to define one or more syntactic groupings and provide *rules* (i.e., logic formulas) and some *actions* (i.e., a block of arbitrary code enclosed with `{: :}`) for assembling them from their constituent elements. One kind of grouping is called a **type expression** [3]. For example, a type expression for a record data type might be described in plain English as: "A record is data that contains other data, which are grouped together into a single value type". Another would be: "A record is a value type that encapsulates data and related functionality". The essential structure of a record can be deduced from either of these descriptions as follows (where `?` means optional, and `*` means 0-or-more):

$$\textit{Record}\ \langle \textit{name} \rangle\ \texttt{\{}\ \langle \textit{field-decls} \rangle^*\ \texttt{\};}$$

where ⟨*field-decls*⟩ is a declaration of the form

$$\langle \textit{type} \rangle\ \langle \textit{name} \rangle\ \texttt{;}$$

or a declaration of the form

$$\langle \textit{type} \rangle\ \langle \textit{name} \rangle\ \texttt{(}\ (\langle \textit{type} \rangle\ \langle \textit{name} \rangle\ (\texttt{,}\ \langle \textit{type} \rangle\ \langle \textit{name} \rangle)^*)^?\ \texttt{);}$$

Of course, this representation can be broken down into simpler terms, but that is a lexicographic issue rather than a grammar issue. The expression, as well as the field or the function declaration, are syntactic groupings in the aforementioned record. Although the full grammar for a *Record* type may include dozens of additional language constructs (and thus production rules) in a language like C#, each with its own nonterminal symbols, we utilize a common concept that involves *naming* (sequence of characters), *repetition* (+ or *),

alternatives (`?`), *order-independence* (`( )`, `{ }`, or `[ ]`), and *value ranges* (`...`) to describe the semantics of the above notation. These symbols will form the foundation of type expressions' footprints[45] and signatures (see Section 5.3.4 on page 122).

Another kind of grouping is called a **predicate** [5], and it plays a big role in *Jiapi*, as a user-defined or internally implemented. The user will typically have to specify a limited collection of predicates and functions, as well as some *true/false* groundings. For example, the ability to determine if a given type is a specified type or to throw some form of exception is rather useful in the development of a type checker. These tasks can be accomplished using a number of semantic predicates, which are conditions that must be met at compile-time before creating an executable file; therefore, a semantic predicate can be defined and implemented as needed. There are two types of semantic predicates that we distinguish: (1) *type predicates* for each atomic type and constructed type, and (2) *validated predicates* for throwing exceptions if their conditions are not met.

A predicate expression evaluates to either *true* or *false*. As an example of a type predicate, consider the following:

$$Record_?(\tau) = \tau == X$$

where X is the type *Record* as defined in the type system of the target language. On the other hand, a validated predicate is simply a block of code followed by a question mark operator (`?`), where ... can be either a simple Java statement or a type predicate:

$$\{...\}?$$

For example, consider the *Record* predicate, which determines if τ is a Record type, embedded in the following validated predicate

$$\{\ Record_?(\tau)\ \}?$$

when the generated code for this predicate executes, it will throw an exception if and only if τ is *not* a type *Record*. As may be expected, we can catch the exception in an exception handler before displaying the appropriate error message.

[45] A footprint of any type τ in an expression e is a collection of symbols that are used to represent τ occurrences within e.

In many object-oriented languages, atomic types are commonly implemented in a single file, as a class (e.g., *PrimitiveType*) that may subclass an abstract $Type$, with several constant declarations, where each represents a separate atomic type (e.g., `INT_TYPE`, `DOUBLE_TYPE` , etc.). An example of a type predicate for an atomic type in Espresso with an embedded action is shown below:

$$Integer_?(X) = \ \{: X.type == PrimitiveType.\texttt{INT\_TYPE} :\}$$

where X is, in Espresso, a parse-tree node type and *type* is an integer value that represents one of the constants declarations. Again, an action is a block of text written in the target language and enclosed in curly brackets. The main difference between the definition of $Record_?$ and $Integer_?$ is that, for the latter, the code enclosed in `{:` and `}` is copied verbatim into the generated code.

The last kind of grouping is a **logic formula**. The essential idea is that a type can be thought of as a proposition, and a value can be thought of as a proof of the statement that corresponds to its type. Thus, we can combine these syntactic groupings with a particular number of terms to form a semantic model (see Section 2.2 on page 20), where we consider the problem of representing and reasoning about type checking using set notation, and first-order logic with quantification over types. For example, consider the following definition for two record types, where ... is a field declaration:

$$r_1 = Record(name_1, (...))$$
$$r_2 = Record(name_2, (...))$$

If we wanted to incorporate the check that r_1 and r_2 are both records, we could write

$$(r_1 =_{\boldsymbol{\tau}} r_2) \Leftrightarrow Record_?(r_1) \wedge Record_?(r_2) \wedge (name_1 = name_2)$$

where $Record_?$ takes a type (as defined in the type system) and returns *true* if the *type* is an instance of a *Record* type and was declared using the same name (i.e., the name of the record). However, if we wanted our type checker to consider the structure of a record rather than just its name, we could instead write

$$(r_1 =_{\boldsymbol{\tau}} r_2) \Leftrightarrow Record_?(r_1) \wedge Record_?(r_2) \wedge (n_1 = n_2) \wedge \big(\bigwedge_{i=1}^{n_1} f_{1,i} \sim_{\boldsymbol{\tau}} f_{2,i} \big)$$

which says that, in addition to our first definition, record types must have the same number of fields (i.e., $n_1 = n_2$), and the i^{th} field in r_1 has to be type equivalent with the i^{th} field in r_2, and vice versa.

5.3.1 Stages in *Jiapi*

The type system design process using *Jiapi*, from grammar specification to a working type checker, has these parts to it:

1. Formally specify the types along with the set of rules for recognizing types with expressions.
2. For each type in the language, describe the action that is to be taken when an instance of the type is recognized. The action is described using predicates, set notation and first-order logic.
3. Write error-reporting routines.

We must follow these steps to turn this source code into a runnable program:

1. Run *Jiapi* on the type system specification file in order to produce the type checker.
2. Import the generated `*.java` file(s) for compilation.
3. Compile the code output by *Jiapi*, as well as any other required source files.

5.3.2 Structure of a *Jiapi* File

A Java *Jiapi* file has at least four parts to it: A **header** section (a header followed by an optional options and declarations), a **types** section, a **clauses** section, and a **type checker** specification section. The structure of the file is as follows:

```
{header, options and declarations}
%%
{atomic and constructed types}
%%
```

```
{clauses}
%%
{type checker}
```

It should be said that the file containing these sections must end with `.jiapi`. The `%%` is a punctuation that appears in every *Jiapi* file to separate sections.

5.3.2.1 Header Section

This is the first section in the file. It includes source code that must come before any generated code, such as a package and import statements. This is used to indicate the package for the generated code, any imported classes, the file name, and the target language, all of which can appear in any order. A header section could look like this:

```
file test (1)
package demo; (2)
use foo; (3)
use bar.baz.foo; (4)
use bar.baz.*; (5)
use bar.baz{a as num1, b as num2} (6)
```

After these declarations, there is a series of optional declarations that allow user code to be inserted in the generated file. Next in the specification is the optional `code requires` declaration which has the form:

```
code requires {: (7)
  ... (8)
:}
```

This declaration permits the inclusion of code directly within the generated file. Let us now go through the part of the header line-by-line:

- `file test` (1) tells *Jiapi* that the type checker is called `test` and thus should be located in a file called `test.java`.

- A package functions in the same way as it does in Java. The line `package demo` ❷ specifies the location of the file and where the generated Java files should be placed. Since *Jiapi* generates code in that same `demo` directory where this file is, the Java compiler will anticipate classes in package `demo` to be in directory `demo`.

- An import statement can be used to import both a local file and a library file. For example, the `foo` class is imported by the line `use foo;`, which is presumed to be in the same directory `demo` ❷. A basic import statement, which fully defines the class name as well as the package with absolute path, can be given if necessary, such as `use bar.baz.foo;` ❹. This will import the file `foo` from the `bar.baz` package located in a directory `bar/baz`. However, by specifying the file name as an asterisk (i.e., `*`), we can import all files in a package, such as `import bar.baz.*;` ❺. We can also provide alternative names for existing types (e.g., `a as num1, b as num2` ❻). This instructs the tool to use whatever defined alias there is or introduce a different shorter name for names that are too long.

- `code requires` ❼ is the optional section in the header (i.e., it does not need to be present in the file), as it consists of optional code fragments. This is where we define Java code to be included within the generated code.

  Note that it is only permissible to declare valid data types as defined by the implementation language. Alternatively, an import statement in the header section should be used to import user-defined data types.

Figure 23 lists the context-free grammar for a header declaration.

$$
\begin{array}{rcl}
\langle \textit{prologue} \rangle & \rightarrow & \langle \textit{file} \rangle\ \langle \textit{package} \rangle\ \langle \textit{imports} \rangle\ \langle \textit{code} \rangle^{?} \\
\langle \textit{file} \rangle & \rightarrow & \textbf{file}\ \texttt{ID} \\
\langle \textit{package} \rangle & \rightarrow & \textbf{package}\ \texttt{ID} \\
\langle \textit{imports} \rangle & \rightarrow & \textbf{use static}^{?}\ \langle \textit{qualified\_name} \rangle \\
 & | & \textbf{use static}^{?}\ \langle \textit{qualified\_name} \rangle\ \texttt{.}\ \texttt{*} \\
 & | & \textbf{use static}^{?}\ \langle \textit{qualified\_name} \rangle\ \texttt{\{}\ \langle \textit{aliases} \rangle\ \texttt{\}} \\
\langle \textit{qualified\_name} \rangle & \rightarrow & \texttt{ID (.ID)}^{*} \\
\langle \textit{aliases} \rangle & \rightarrow & \texttt{ID (}\ \textbf{as}\ \texttt{ID)}^{?} \\
\langle \textit{code} \rangle & \rightarrow & \textbf{code requires \{:}\ \langle \textit{user\_code} \rangle\ \textbf{:\}}
\end{array}
$$

Figure 23. Context-free grammar for a header declaration.

5.3.2.2 Types Section

The first mandatory section of the specifications follows the user-supplied code: A lists of types. These declarations are responsible for giving each type in the language a name and a type, with each type being represented by a parse-tree node type. A types section could look like this:

```
atomic { ❶
  boolean, ❷
  void, ❸
  [byte < short < int < long < float < double], ❹
  [char < int] ❺
}

constr_1 ... ❻
      ⋮
constr_n ... ❼
```

Let us take a look at each line individually:

- `atomic` ❶ is used to group and create a type hierarchy (or type lattice). It is not necessary for the hierarchy to be linear as it could branch; a branch is created after a type (or group) is followed by a comma (`,`). For example, from this grouping, *Jiapi* creates a directed graph and returns a two-dimensional array of nodes, where each row and column represents a primitive type that might appear before all nodes to which

it points. In this lattice or hierarchy, some of these types might be *ranked* using the ordering operator (or $<_\tau$) and according to the group to which they belong, resulting in a total ordering of the types; this is depicted in Figure 24.

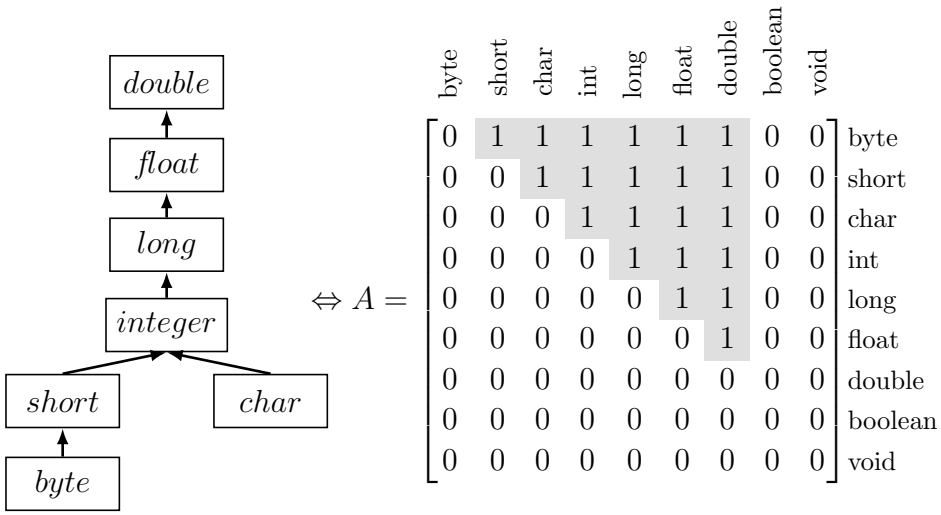


Figure 24. Two-dimensional array of nodes.

- Each of the types `boolean` ❷ and `void` ❸ represent a sub-tree of a node in the type hierarchy and, therefore, they are not part of any group and exist independently (as seen in Fig. 13 on page 77).

- A pair of square brackets (`[ ]`) is used to create not only new groups of numeric types within the atomic group but also atomic types without $<_\tau$ that are not numeric, as shown in ❹ and ❺.

- `constr ...` ❻ is the declaration of a pattern that will be used to type check constructed types. It is to be noted that, whereas atomic types can only ever have one grouping, constructed types can have multiple declarations in the types section ❼.

Figure 25 lists the context-free grammar for creating a grouping of atomic types.

$$
\begin{array}{rcl}
\langle types \rangle & \rightarrow & \langle primiti\_types \rangle^{?}\ \langle constructed \rangle^{*} \\
\langle primiti\_types \rangle & \rightarrow & \textbf{atomic}\ \texttt{\{}\ \langle atomics \rangle\ \texttt{\}} \\
\langle atomics \rangle & \rightarrow & \langle atomic \rangle\ (\texttt{,}\ \langle atomic \rangle)^{*} \\
\langle atomic \rangle & \rightarrow & \texttt{ID}
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \langle atomic\_list \rangle \\
\langle atomic\_list \rangle & \rightarrow & \texttt{[}\ \langle elements \rangle\ \texttt{]} \\
\langle elements \rangle & \rightarrow & \langle atomic\_type \rangle\ (\texttt{<}\ \langle elements \rangle)^* \\
\langle atomic\_type \rangle & \rightarrow & \texttt{ID}
\end{array}
$$

Figure 25. Context-free grammar for atomic types.

5.3.2.3 Clauses Section

This section consists of declarations and logical statements, which often incorporate variables that span sets of possible instances, referred to as universes. It is also – and **exclusively** – used to create operators and operations that pertain to a node type. These operators and operations can be re-implemented or use in other clauses. We use quantifiers to specify that something holds for all possible instances or for some but possibly not all instances. Declarations and statements are constructed from terms using the logical connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies or if-then), and $\Leftrightarrow$ (equivalent to), that can be prefixed by $\forall$ (universal) or $\exists$ (existential) quantifiers, or be prefixed by a $\bigwedge$ (conjunction), $\bigvee$ (disjunction), $\bigcup$ (big-union), or $\bigcap$ (big-intersection) of variables. Other symbols, such as $\in$ (in), $\subset$ (subset), $\setminus$ (set-minus), $\cup$ (union), $\cap$ (intersection), $\emptyset$ (empty set), etc., from set theory may also be used (see Table 8 on page 109 for more symbols).

A clauses section could look like this:

```
clause Class { ❷
  def Class?(t)= {: t.name == X :} ❶
  def t1 =T t2 <=> Class?(t1)/\ Class?(t2)/\ ❸
   t1.name = t2.name
  def t1 ~T t2 <=> t1 =T t2 ❹
  def t1 :=T t2 <=> t2 </_T t1 ❺
  def t1 <:T t2 <=> t1 = t2 \/ t1 <:T super \/ \/ i in 1..n:
   t1 <:T interfaces[i] ❻
  ...
}
```

Let us go over the part of the clauses, line-by-line:

- `clause Class` ❷ is used to group and create a list of operations that pertain to

types in the target language. A *clause* has a name, a number of predicates and functions, a number of local variables introduced in the body, and the body itself. Any local variables used in the body must be "forward-declared"; that is, for local variables, they must either represent the same clause, a constructed type, or one of the members of its constructed type equivalent. For instance, we may have a variable of type *Class*, any constructed type that was previously defined, or one of the atomic types − as defined earlier in the atomic type section − in the body of the above clause. Predicates and functions behave similarly to how they would in a language like Java.

- We define the $=_\tau$ operator in ❸, and use the a predicate in ❶ to determine if two types `t1` and `t2` are classes.
- Using the definitions for `t1` and `t2`, we proceed to define the $\sim_\tau$ operator in ❹ and the $:=_\tau$ operator in ❺.
- Finally, we implement a new operator − the subtyping operator ($\prec_\tau$) − for classes in ❻. Note that the above clause is a description of what, in a language like Java, defines a class as a *class*. Therefore, a clause typically has access to all the components that are relevant to its constructed type equivalent. This means that we access these elements in the same way that we would access members of a class, with the exception that their access modifier is public by default.

Figure 26 lists the context-free grammar for a clause declaration.

$$
\begin{array}{rcl}
\langle \textit{clause} \rangle & \rightarrow & \textbf{clause}\ \texttt{\{}\ \texttt{ID} \mid \textbf{atomic}\ \texttt{)}\ \langle \textit{clause\_body} \rangle^*\ \texttt{\}} \\
\langle \textit{clause\_body} \rangle & \rightarrow & \langle \textit{field\_declaration} \rangle \mid \langle \textit{method\_declaration} \rangle \\
\langle \textit{field\_declaration} \rangle & \rightarrow & \textbf{static}^?\ \texttt{(}\ \textbf{def} \mid \textbf{let}\ \texttt{)}\ \texttt{ID}\ \texttt{(}\ \texttt{:}\ \langle \textit{type} \rangle\ \texttt{)}^?\ \texttt{(}\ \texttt{<-}\ \langle \textit{expression} \rangle\ \texttt{)}^? \\
\langle \textit{method\_declaration} \rangle & \rightarrow & \textbf{def}\ \texttt{ID}\ \texttt{(}\ \langle \textit{params} \rangle^?\ \texttt{)}\ \texttt{(}\ \langle \textit{return\_type} \rangle \mid =\ \texttt{)}\ \langle \textit{expression} \rangle \\
& \mid & \textbf{def}\ \texttt{ID}\ \langle \textit{operator} \rangle\ \texttt{ID}\ \textbf{equival}\ \langle \textit{expression} \rangle
\end{array}
$$

Figure 26. Context-free grammar for a clause declaration.

5.3.2.4 Type Checker

Jiapi permits the user to embed actions in the form of visits for type checking.

```
action ClassType for ct{
  T(ct)!= nil => return T(ct) ❶
  def cd = new Class()
  T(ct)= cd ❷
  return T(ct) ❸
}
```

The action statement above, for example, defines a block of code that is embedded into the type checker visitor. It is clear that action names are referred to by the name of the variable that represents a **syntactic node**, and they are executed when a type computation of the form $\mathcal{T}(...)$ takes place ❶. However, if the type of a node is already known, and if $\mathcal{T}(...)$ appears to the left of an assignment expression ❷, then it becomes a *getter* method; otherwise, it becomes a *setter* method as it appears anywhere as in the return statement ❸. We should emphasize that the placement of action is entirely up to the user due to the compositional nature of the visitor pattern.

Figure 27 lists the context-free grammar for the type checker section.

$$
\begin{array}{rcl}
\langle \textit{type\_checker} \rangle & \rightarrow & \langle \textit{field\_declaration} \rangle \\
& \mid & \langle \textit{action\_declaration} \rangle \\
\langle \textit{action\_declaration} \rangle & \rightarrow & \textbf{action}\ \langle \textit{type} \rangle\ \textbf{for}\ \texttt{ID}\ \langle \textit{return\_type} \rangle^{?}\ \langle \textit{block\_expression} \rangle
\end{array}
$$

Figure 27. Context-free grammar for a clause declaration.

5.3.3 Describing Primitive Types

Any type that a program can use but **cannot** build using type constructor is an atomic (or primitive) type in the language. Numeric types such as *integers* and *doubles*, as well as *booleans*, *characters*, and, in some languages, *strings*, are examples of atomic types, so we can describe them by their names alone. Table 14 summarizes the primitive types in Espresso.

Table 14. Primitive types and their representation.

Atomic Types	
Type	**Representation**
byte	*byte*
short	*short*
char	*char*
integer	*integer*
long	*long*
float	*float*
double	*double*
boolean	*boolean*
void	*void*

5.3.4 Describing Constructed Types

Any type that a program can construct from existing atomic types or other constructed types created using the language is a constructed (or structured) type in the language. Some examples of constructed types are records, classes, interfaces, unions, and enumerations.

Although each language has its own set of types and rules for describing them, we use patterns in order to describe them in a more general manner. In *Jiapi*, patterns are used to match values to structures and, in some cases, to bind variables to values within these structures. They are also utilized in variable declarations and parameters for functions (or methods). Figure 28 shows a pattern made up of just five terms and all possible recursive combinations. (Note, a word in *italic* font represents a "tag"[46].) An example of a recursive data type, for reference types only, is given in Section 5.3.4 on page 124.

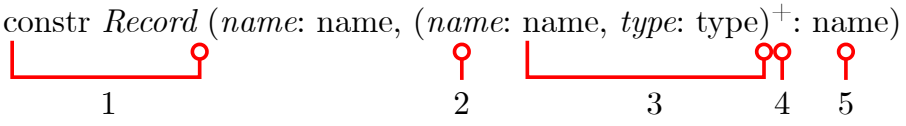


Figure 28. Pattern for constructing a *Record* data type.

The notation in Figure 28 appears to be slightly different from that of a meta-

[46] In *Jiapi*, a *tag* is a reserved word use to distinguish what constitutes a name from a type.

syntax[47] [233, 234], but it is not:

1. Creates a constructed type called *Record*.
2. Assigns a **name** to the constructed type.
3. Creates a collections of values, where each component is identified by a different field **name** and **type**.
4. At least one field will be present in the collection.
5. Assigns a **name** to the collection of values, which is used as an accessor in the generated code.

Using this pattern, we can construct a record named *name* with fields n_i of type t_i for $i \in 1..m$, in order, where a field is declared like a local variable with a type and a name. Note that this pattern (which we shall refer to as *type constructor*) is used to represent a **record data type** during type checking. Thus, from a constructive point of view, a type is either one of a narrow set of built-in types (e.g., integer, character, boolean, etc., which are often known as primitive or predefined types) or a composite type generated by applying a type constructor to one or more simpler types. For example, the following snippet of code demonstrates the *red*, *green*, and *blue* properties (a fixed set of primitive types) of a Color `struct` in C#.

```
struct Color {
  byte red;
  byte green;
  byte blue;
}
```

Here, we verify that the above type matches that expected by its pattern. A *Record* pattern, therefore, matches record values that match all criteria defined by its sub-patterns or terms (e.g., its fields are referenced by *name* and their types by *type* respectively), which are also use to **destructure** (i.e., break up) the record.

[47] Informally speaking, a *meta-syntax* is a shorthand for the phrase "meta-language syntax".

Now, how do we compose a type and defined operations on them? One way to do this is by parameterizing the expression of a type as seen below.

$$Record(name, ((n_1, t_1), ..., (n_m, t_m))) \tag{5.1}$$

One may want to think of $(..., (...))$ as the *signature* of the type constructor. Intuitively, the *Record* type constructor takes a pair consisting of a *name* and a collection of values stored together as one, where each component is identified by a different field name, n_i, and type, t_i. Therefore, the Color `struct` has the following type:

$$Record(Color, ((r, byte), (g, byte), (b, byte)))$$

For convenience, we may use "syntactic sugar" as a shorthand; that is, we may introduce multiple parameters to an expression using the *positive closure* (or +) as a shorthand for $((n_1, t_1), ..., (n_m, t_m))$ only to make notation on paper more clear and examples more comprehensible; therefore,

$$(n, t)\texttt{+} = ((n_1, t_1), ..., (n_m, t_m))$$

Note that this abbreviation has already been used in Figure 28. Let us now move on onto recursive patterns.

A recursive data type is one that can contain other values of the same type as a property [3, 5]. Often, when we want to construct dynamic data structures, we use recursive data types, such as lists or trees, and depending on our run-time needs, the size of these dynamic data structures can expand or shrink. For example, a linked list node has a recursive structure that includes both data and computation, requiring the adoption of a recursive pattern. The snippet of code for a node element, such as

```
struct node {
  int data;
  node next;
}
```

defines the recursive type `node` as a class that contains a field `data` and a field `next` of type `node`. As a result, this linked list can be of the following type:

$$Record(node, ((a, int), (next, node)))$$

Note that one of the expressions, namely $(next, node)$, is recursive: It is defined in terms of a *Record* type. Recursion naturally captures the tree structures found in programming language syntax. For example, we can expand the expression *next* (which is demonstrated by thin red dashed lines) to show how a *Record* type is structured recursively. Here, subscripts are used to keep track of multiple occurrences of the same type expression in Figure 29.

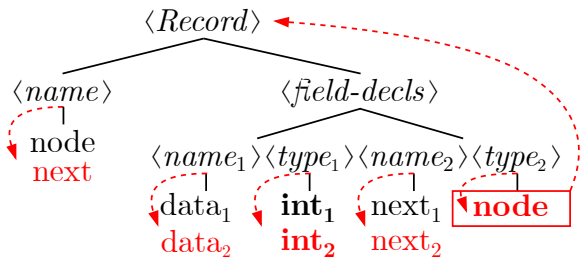


Figure 29. Recursive type expression.

As a common rule, when analyzing the structure of a data type, it is important to remember that we are interested in the **abstract syntax** of its type constructor (e.g., Pattern 5.1) rather than the **concrete syntax** of the type (e.g., the correct syntax for writing a `struct` in C#). In such case, we prefer to think of type constructors as *abstract syntax trees*, and it should be obvious that brackets (`( )`, `[ ]`, or `{ }`) are simply a means to describe these two-dimensional tree-like data structures as a linear character of strings. The goal is to pair each language construct with an expression that describes its type but without adding a new form of expression to the abstract syntax. This is done so that a type constructor can define type expressions inductively from fundamental − or basic − types and constants. An object-oriented approach to dealing with type constructors proceeds as follows:

1. Identify the components of the constructed type.

2. Encapsulate those components into a class.

3. If necessary, include "hooks" in the class to access needed information stored in a different location.

Figure 30 depicts the entire scenario. The *visitor pattern* [63, 64] is then applied to define the actions for operating on the type, which allows us to control when certain actions are performed: On the one side, we have data access, which understands how to traverse a data structure, while on the other, we are focused on a specific computation that needs to be executed. Whether it is a pre-order, an in-order, or a post-order traversal, or a combination, this can also be done by hand after extending the visitor pattern. The end result is a reusable and extendable system of cooperating objects that is also simple to understand and maintain.

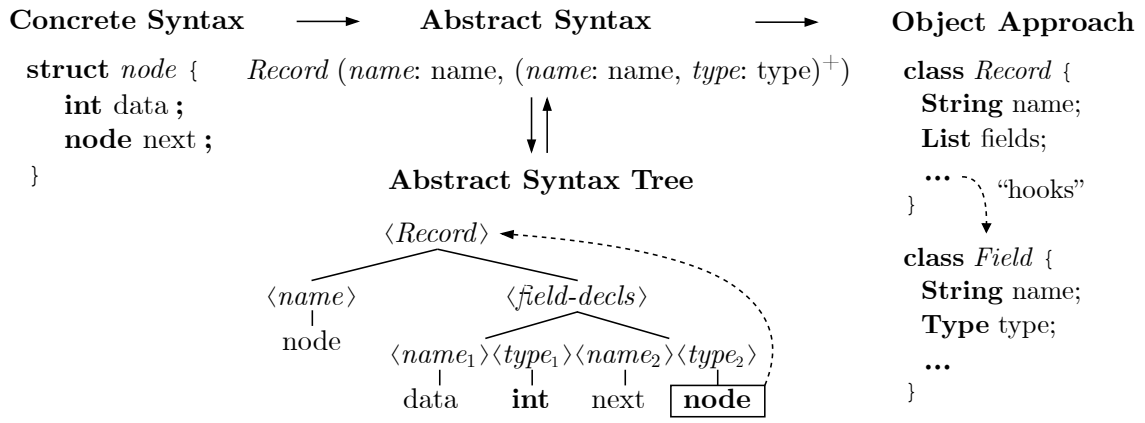


Figure 30. Transformation from concrete syntax to abstract syntax tree to object-oriented approach.

5.3.5 Patterns for Constructed Types

A constructed type can have several patterns listed, each having different elements used to described it. In simple cases, it is sufficient enough to provide:

1. A line that declares the name of a constructed type, such as

$$\text{constr } \underbrace{\textit{Record}}_{\text{name}} (...)$$

2. A list of names and types, such as

$$\text{constr } ... \underbrace{((n_1, t_1), ..., (n_m, t_m))}_{\substack{\text{list of names} \\ \text{and types}}}$$

   with fields name n_i of type t_i for $i \in 1..m$

3. A type followed by a list of names, such as

$$\text{constr } ... \ (t, \underbrace{(n_1, ..., n_m)}_{\text{list of names}})$$

with fields name n_i of type t for $i \in 1..m$

4. A meta-character that precedes a name, a type, or follows a list (a sequence of elements), such as

$$\text{constr } ... \ (\underbrace{(n, t)}_{\substack{\text{list of names} \\ \text{and types}}}\overbrace{+}^{\substack{\text{positive} \\ \text{closure}}}) \quad \text{or} \quad \text{constr } ... \ (\underbrace{t}_{\substack{\text{same} \\ \text{type}}}, \underbrace{(n)}_{\substack{\text{list of} \\ \text{names}}}\overbrace{+}^{\substack{\text{positive} \\ \text{closure}}})$$

5. A range operator that represents a set of elements greater or equal to δ or less or equal to η, where δ and η are placeholders for real values, such as

$$\text{constr } ... \ (..., [\delta..\eta])$$

A range of values can be matched using the binary operator `[ .. ]`. The expression `[`δ`..`η`]` matches characters in that range inclusively, and only the lower, δ, and upper, η, bounds are stored in the implementation of this operator, making it lightweight. We use the range operator to determine the length of an array or list; for example, for a declaration

```
int a[5];
```

we can use the range operator as follows:

$$\text{constr } \underbrace{Array}_{\text{name}} \ (\underbrace{int}_{\text{type}}, \overbrace{[0..5]}^{\substack{\text{range} \\ \text{operator}}}) \quad \text{or} \quad \text{constr } \underbrace{Array}_{\text{name}} \ (\underbrace{int}_{\text{type}}, \overbrace{[\,]}^{\substack{\text{empty} \\ \text{range}}})$$

As we shall see later, these patterns are used to **destructure** a constructed type into its constituent elements during code generation, thus making static type-checking easier. The syntax used is almost the same as when creating such types. Figure 31 lists the context-free grammar for creating constructed types.

$$
\begin{array}{rcl}
\langle \textit{constructed} \rangle & \rightarrow & \texttt{ID}\ \langle \textit{contructed\_list} \rangle \\
\langle \textit{contructed\_list} \rangle & \rightarrow & \texttt{(}\ \langle \textit{constructed\_element} \rangle^{*}\ \texttt{)} \\
 & \mid & \texttt{\{}\ \langle \textit{constructed\_element} \rangle^{*}\ \texttt{\}} \\
 & \mid & \texttt{[}\ \langle \textit{constructed\_element} \rangle^{*}\ \texttt{]} \\
\langle \textit{constructed\_element} \rangle & \rightarrow & \texttt{ID}\ \texttt{(}\ + \mid * \mid \texttt{?}\ \texttt{)}^{?}\ \texttt{:}\ \textbf{id} \\
 & \mid & \texttt{ID}\ \texttt{(}\ + \mid * \mid \texttt{?}\ \texttt{)}^{?}\ \texttt{:}\ \texttt{ID} \\
 & \mid & \texttt{ID}\ \texttt{(}\ + \mid * \mid \texttt{?}\ \texttt{)}^{?}\ \texttt{:}\ \textbf{type} \\
 & \mid & \texttt{ID}\ \langle \textit{range\_inclusive} \rangle\ \texttt{ID}\ \texttt{:}\ \texttt{ID} \\
 & \mid & \langle \textit{constructed\_list} \rangle\ \texttt{(}\ + \mid * \mid \texttt{?}\ \texttt{)}^{?}\ \texttt{:}\ \texttt{ID}
\end{array}
$$

Figure 31. Context-free grammar for a constructed type declaration.

5.3.5.1 Sequence of ⟨*constructed_list*⟩

Recall, constructed types define data structures that contain various types of data members (e.g., constants and fields, such as methods, properties, operators, etc.) that may need to be type checked in a number of ways. The big question here is: How should these data members be traversed when writing a source-to-source translator for a language? The idea is to decouple some behavior from a collection of data and the structure that contains them. Typically, a data set could have an *accept()* method that iterates over all values and later calls a *visitor.visit()* method for each value [5], allowing the visitor to see all of the values. Unfortunately, the standard visitor pattern is incapable of traversing a hierarchical structure[48], leaving us with only two options: (1) use an alternate method of traversal or (2) search all branches with no chance of matching.

We will use the former approach in this thesis to allow operations to be performed on the nodes of a hierarchical object structure, as explained in Example 8.

Example 8 (Constructed Type: Record): As seen before, atomic types equivalence is usually straightforward to establish; for example, an *int* is only equivalent or equal to

[48] It does not support traversal depth tracking or short-circuit branch traversal [235].

another *int*, a *double* is only equivalent to another *double*, etc. The ability to determine the equivalence of constructed types is also required in many languages. When dealing with constructed types, however, one typical strategy is to store the essential information defining the type in tree structures. Consider the same recursive type `node` (from Section 5.3.4 on page 124) but with one additional field (*prev*):

```
struct node {
  int data;
  node next;
  node prev;
}
```

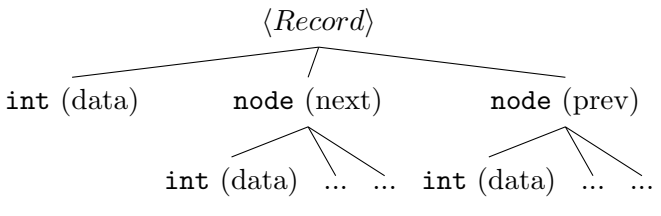


Figure 32. Basic information defined as a tree structure.

Here is a set of rules for building such tree:

Type	**Description**
record	one subtree for each field
atomic	one subtree
reference	one subtree containing the type that the pointer refers to

Checking the equivalence of two types becomes a straightforward recursive tree operation if we store the type information in this way. The following is an outline of a recursive structural equivalence test:

Input: Types α and β
Output: *true* if the types are structurally equivalent

```
1  Function structuralEquivalence
2      if α and β are struct types ❶ then
3          if |α| ≠ |β| then
4              return false
5          result = true
6          ∀t_i : ∀t_j : t_i ∈ α ∧ t_j ∈ β: result &= structuralEquivalence(t_i, t_j)
7          return flag
8      if α and β are primitive types ❷ then
9          t_α = α as primitive type
10         t_β = β as primitive type
11         return t_α == t_β
12     if α and β are reference types then
13         return α == β
14     return false
```

Algorithm 1: A simple visitor pattern for structural equivalence.

The method is rather simple, but is it guaranteed to terminate? Our record type is recursive (i.e., it contains a reference to a record of the same type), so we need to be careful when traversing the tree in ❶; otherwise, the visit invocations in ❷ will never execute. To get around this, we could "mark" the tree nodes as we traverse the tree, allowing us to detect cycles and limit equivalence on reference types. Doing this would require keeping track of when we enter and exit a constructed type, with each call of *accept()* returning a boolean traversal status for the tree's depth. For example, if *accept* on a constructed type answers *true*, the traversal immediately stops at that tree depth. □

The standard visitor can *only* tell us when we are entering a node (i.e., a constructed type). We have no idea if we exited the prior node before entering the current one; therefore, we cannot track when we are entering and leaving a node. Since we cannot implement this strategy using the traditional visitor pattern, we device the following patterns to allow conditional navigation. These patterns will enable us to skip unnecessary branches and all of their children.

5.3.5.2 Elements and Patterns

Each element and language construct pertaining to *Jiapi* is described below. We use the character α, γ, or β to refer to an element without a *tag*, and use the term *group* to describe the sequence of elements α, γ, and β enclosed in one of these: `()`, `[]`, or `{}`. Whenever needed, and for the remaining of this chapter, we will use $[\![\,]\!]$ to indicate that either `()`, `[]`, or `{}` may be used, except where noted. We also use the $\oplus$ operator to indicate that either `?`, `+`, or `*` may be used, except where noted.

Elements

An element is either a name preceded by a *tag*, which can be one of these:

- **name:** *name*
- **type:** *name*

or a group enclosed in brackets followed by an optional closure operator, and an optional colon and name, such as:

- $[\![...]\!] \oplus$ **:** *name*

Patterns

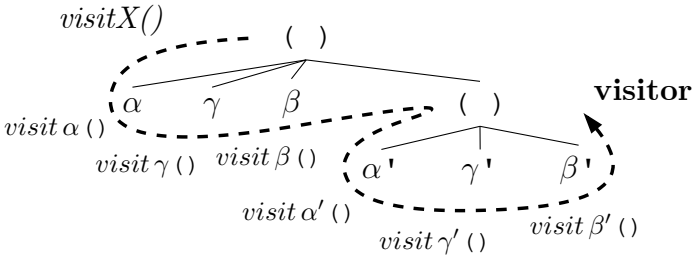
- **Pattern #1:** A group enclosed with *round* brackets (`()`) represent an ordered sequence of elements. This guarantees iteration order over each of their elements, which is normally the order in which elements were inserted into the sequence (insertion-order), provided that the sequence is not structurally modified at any time (i.e., we do not make any insertions while traversing a sequence of elements). Note that the symbol $\rightarrow_i$ (meaning left-to-right) represents the order traversal of elements in a sequence of steps i. For example, the sequence

  `(`α γ β `(`α`'` γ`'` β`'))`

  can be traversed as follows

1. $\rightarrow_1\ \alpha\ \rightarrow_2\ \gamma\ \rightarrow_3\ \beta\ \rightarrow_4\ (\rightarrow_5\ \alpha'\ \rightarrow_6\ \gamma'\ \rightarrow_7\ \beta')$

which means "match α γ and β sequentially", then "match α' γ' and β' sequentially". Here is the familiar visitor pattern operating on this sequence in a depth-first walk.



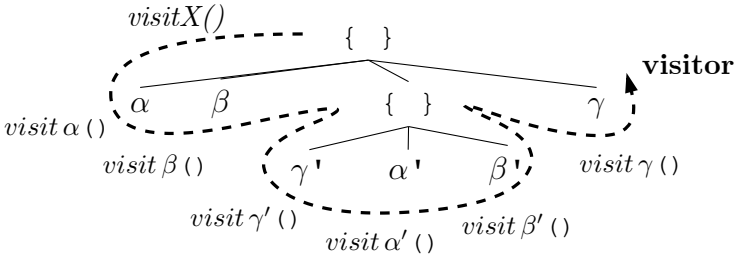
- **Pattern #2:** A group enclosed with *curly* brackets ({ }) represent an unordered sequence of elements. Since elements in an unordered sequence are just that, *unordered*, it does not matter whether groups are enclosed in the same or different pairs of brackets. Put differently, there is no first, last, or, in general, *i-th* element in the sequence. For example, the sequence

  $\{\alpha\ \gamma\ \beta\ \{\alpha'\ \gamma'\ \beta'\}\}$

  can be traversed as follows

  1. $\rightarrow_1\ \alpha\ \rightarrow_2\ \beta\ \rightarrow_3\ \{\rightarrow_4\ \gamma'\ \rightarrow_5\ \alpha'\ \rightarrow_6\ \beta'\}\ \rightarrow_7\ \gamma$

  The diagram shown below shows a "random" walk, represented by the thick dashed line.



  It also can be traversed in two other ways, such as

(b) $\rightarrow_1$ {$\rightarrow_2$ β' $\rightarrow_3$ γ' $\rightarrow_4$ α'} $\rightarrow_5$ γ $\rightarrow_6$ β $\rightarrow_7$ α

(c) $\rightarrow_1$ γ $\rightarrow_2$ {$\rightarrow_3$ β' $\rightarrow_4$ α' $\rightarrow_5$ γ'} $\rightarrow_6$ α $\rightarrow_7$ β

Although there are several ways to traversed the above sequence, as it is unordered, the element order should be implicit in the syntax of the constructed type; that is, the *tree* structure that should be produced from the constructed type is traversed in a left-to-right depth-first order.

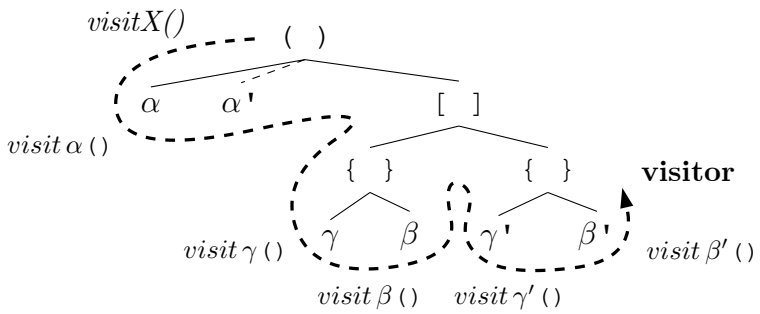
- **Pattern #3:** A group enclosed with *square* brackets (`[ ]`) represent a required sequence of elements. By default, only one element is required if included in `( )` or `{ }`. This is because not all elements in a constructed type might be needed (e.g., a class may or may not extend another class or a group of classes, or may or may not contain fields). However, it is sometimes necessary to "explicitly" labeled words as **required** using square brackets (i.e., if we required a class to have fields then we make it so by enclosing an element − or a group − in `[ ]`). For example, when grouping mutually-exclusive elements in `( )` or in `{ }`, like in the following sequence

  (α α' [{γ β}$_1$ {γ' β'}$_2$])

  only **one** element in the sequence (α α'...) (i.e., α or α'), and in the first sequence of {γ β} (i.e., γ or β) and second sequence of {γ' β'} (i.e., γ' or β') would be required. The entire sequence can then be traversed as follows

  1. $\rightarrow_1$ α $\rightarrow_2$ [$\rightarrow_3$ {$\rightarrow_4$ γ $\rightarrow_5$ β} $\rightarrow_6$ {$\rightarrow_7$ γ' $\rightarrow_8$ β'} $\rightarrow_9$]

  The diagram shown below represents a "random" walk by the thick dashed line. In addition, although α' is present, it is assumed to be absent from the tree and is represented by a thin dashed line.



It can also be walked in a number of different ways, including

(b) $\rightarrow_1\ \alpha'\ \rightarrow_2\ [\rightarrow_3\ \{\rightarrow_4\ \gamma\ \rightarrow_5\ \beta\}\ \rightarrow_6\ \{\rightarrow_7\ \gamma'\ \rightarrow_8\ \beta'\}\ \rightarrow_9]$

(c) $\rightarrow_1\ \alpha\ \rightarrow_2\ [\rightarrow_3\ \{\rightarrow_4\ \gamma\}\ \rightarrow_5\ \{\rightarrow_6\ \beta'\}\ \rightarrow_7]$

(d) $\rightarrow_1\ \alpha'\ \rightarrow_2\ [\rightarrow_3\ \{\rightarrow_4\ \beta\}\ \rightarrow_5\ \{\rightarrow_6\ \gamma'\}\ \rightarrow_7]$

(e) $\rightarrow_1\ [\rightarrow_2\ \{\rightarrow_3\ \gamma\ \rightarrow_4\ \beta\}\ \rightarrow_5\ \{\rightarrow_6\ \gamma'\ \rightarrow_7\ \beta'\}\ \rightarrow_8]$

Note that groups enclosed in square brackets will **always** appear in any given sequence as they are required; therefore, both $\{...\}_1$ and $\{...\}_2$ are required.

These elements and language constructs, when combined, form valid patterns, each starting with a **keyword** followed by a constructed type name. For example,

constr *name* $[\![...]\!]$

creates a constructed type named *name* consisting of elements (1), and either pattern (1), pattern (2), pattern (3), or a combination of them inside brackets ($[\![...]\!]$). We will put some of these patterns to use next.

5.4 Representing ProcessJ protocols in *Jiapi*

Early on in the chapter, we used a few of these patterns to describe the structure of a record type. Let us now apply these patterns again to define a construct that is more complex, such as that of a *protocol* in ProcessJ [11]. Protocols are comparable to unions in C, except that they are used to specify the structure of a single message sent over a channel [21, see

Section 6.2.1]. A protocol is a type with one or more elements indexed by a tag-name, which is a list of variables preceded by their data types, separated by semicolons, and enclosed by curly brackets. The following is a general layout for a variant protocol:

$$\textit{Protocol}\ \langle \textit{name} \rangle\ (\textit{extends}\ \textit{name}\ (\texttt{,}\ \textit{name})^*\ )\ \texttt{\{}\ \langle \textit{tag-decls} \rangle\texttt{+ \};}$$

where $\langle$*tag-decls*$\rangle$ is a declaration of the form

$$\textit{tag} : \texttt{\{}\ (\textit{fields}\texttt{;})^*\ \texttt{\}}$$

A example variant protocol can be defined as follows:

```
protocol Alternatives {
  dog: { int i; }
  car: { string s; double d; }
  pig: { Packet p; }
  canary: { Message m; }
  poison: { }
}
```

where `Packet` and `Message` are some named types. From the layout and definition, we conclude that a protocol must have at least one *tag-name*, and this tag-name may or may not be followed by an empty pair of `{ }`. We can create a type constructor for such protocol using the preceding patterns, where `[ ]` denotes that anything in between `[` and `]` is required, and everything else is optional. So, for Figure 33

$$\textit{Protocol}(n, \overbrace{((tag_1, \{(n_{1,1}, t_{1,1}), ..., (n_{1,m_1}, t_{1,m_1})\}),}^{\text{An exception is thrown if the set is empty}}$$
$$(tag_2, \{(n_{2,1}, t_{2,1}), ..., (n_{2,m_2}, t_{2,m_2})\}),$$
$$...,$$
$$(tag_k, \underbrace{\{(n_{k,1}, t_{k,1}), ..., (n_{k,m_k}, t_{k,m_k})\}}_{\text{Optional set, no exception is thrown}}))),$$

} At least one element/term is required

Figure 33. Decomposition of a ProcessJ Protocol.

the corresponding type expression is given in Figure 34.

constr *Protocol*(*name*: name, [*name*: name (*name*: name, *type*: type)$^{+}$: fields]$^{+}$:tags)

Figure 34. Patterns for constructing a *Protocol* data type.

Ordering Operator ($<_{\tau}$)

We can also define the order operator ($<_{\tau}$) for protocols in the same way that we did for atomic types using the same approach as we did for Espresso's constructs. To begin, we must first understand the difference between object inheritance in a language like Espresso and protocol inheritance in ProcessJ. Let us consider object inheritance first. Assume we have the following code

```
class A {
  int a;
}

class B extends A {
  int b;
}
```

then a variable of type `A` can hold a reference to an object of type `B`; however, a variable of type `B` cannot hold a reference to a variable of type `A`. To understand why assume that a variable of type `A` could hold a reference to an object of type `A`, and consider the following code

```
...
B tmp = new A();
tmp.b = 100;
```

Clearly, the `b` field does not exists in the `A` object. Now consider the following ProcessJ declarations:

```
protocol A {
  a1: { ... }
  a2: { ... }
}

protocol B extends A {
  b1: { ... }
  b2: { ... }
  b3: { ... }
}
```

We encounter the same issue as we did with object if we are not careful. For example, if the following code was legal we would have a problem:

```
B b = ...
... b.b3 ...
```

The problem here consists of the unchecked access to the variant cases of `b`. In ProcessJ, all access to protocol variables must happen in a *tag-guarded* switch statement. This way the compiler can assure that the wrong fields are never accessed. Since inheritance for protocols in ProcessJ operates in the opposite direction to that of Espresso's, we can say that one protocol type is less than another if its cases are a subset of the other − that is, we take both names and types into consideration. So for two general (protocol) types p_1 and p_2, we have:

$$(p_1 <_\tau p_2) \Leftrightarrow (p_1 \prec_\tau p_2) \vee (\exists \mathcal{X} : \mathcal{X} \prec_\tau p_2) \wedge (\mathcal{X} \prec_\tau p_1)$$

where $p_1 \prec_\tau p_2$ means that p_1 extends p_2; namely, if p_1 either directly or transitively extends p_2. The last part comes from observing that $\prec_\tau$ applied transitively gives $(\mathcal{X} \prec_\tau^* p_1)$, meaning that $p_1 =_\tau \mathcal{X}$ and which further gives us $\mathcal{X} <_\tau p_1$. Trivially $\leq_\tau$ is defined as

$$(p_1 \leq_\tau p_2) \Leftrightarrow (p_1 =_\tau t_2) \vee (p_1 <_\tau t_2)$$

assuming that p_1 and p_2 are both already protocol types. Note that if needed, the ceiling function can be applied to protocol types as well.

Type Equality ($=_\tau$) and Type Equivalence ($\sim_\tau$)

We can also express the $=_\tau$ operator for protocols in ProcessJ in terms of *name equivalence* thanks to what we learned in Chapter 4. For two such types to be the same, they have to be the same type name. Therefore, for two protocols types $p_1 = Protocol(n_1, (...))$ and $p_2 = Protocol(n_2, (...))$, we obtain

$$(p_1 =_\tau p_2) \Leftrightarrow Protocol?(p_1) \wedge Protocol?(p_2) \wedge (n_1 = n_2)$$

Note that type equivalence is equal to type equality so, therefore, for two protocol types p_1 and p_2, we have:

$$(p_1 \sim_\tau p_2) \Leftrightarrow (p_1 =_\tau p_2)$$

Assignment Compatibility ($:=_\tau$)

Last but important, assignment compatibility of protocol types is also straightforward, as discussed when we defined the ordering operator. Thus, for two protocols p_1 and p_2, we have:

$$(p_1 :=_\tau p_2) \Leftrightarrow Protocol?(p_1) \wedge Protocol?(p_2) \wedge (p_1 \leq_\tau p_2)$$

In a similar manner, we could define a type ordering of record types, but we will leave that as an exercise for the reader.

5.5 Back to Record Types

Now that we have returned to records, we will make use of one as a small case study to demonstrate the representation of simple typed λ-calculus rules in *Jiapi* syntax. Recall, *Records* (or structs) are a stylized form of tuples that can be accessed via mnemonic field names. Both are *product types* that can be generalized in a straightforward manner to n-ary products [236]. For example, $\langle 4, \text{“}jiapi\text{”}, true\rangle$ is a 3-ary tuple with a type $int \times string \times boolean$ because it contains an *integer*, a *string*, and a *boolean* value. A record type can therefore be written as

$$\{\ell_i : \tau_i, ..., \ell_n : \tau_n\}$$

for $i \in 1..n$, where $\ell_1, ..., \ell_n$ are distinct field names (or identifiers) and $\tau_1, ..., \tau_n$ are types. This record type is essentially a notational variant of the product type $\tau_1 \times ... \times \tau_n$, although the main difference is that records have *subtyping property*. It is worth noting that the notation used above is equivalent to

$$\{\overline{\ell_i : \tau_i}\}$$

where $\overline{\ell_i : \tau_i} = \ell_1 : \tau_1, ..., \ell_n : \tau_n$.

Let us now write part of the grammar for a record value and its type, as shown in Figure 35. This is a slightly different variation of [114, Figure 11-7] and [237]:

Extensions:

record	e	$=$	$... \{\overline{\ell_i = e_i}\}$
field access		$\mid$	$e.n$
record value	v	$=$	$... \{\overline{\ell_i = v_i}\}$
record type	τ	$=$	$... \{\overline{\ell_i : \tau_i}\}$

Figure 35. Grammar for a record type.

where $\ell_1, ..., \ell_n$ are *labels* and $e_1, ..., e_n$ are corresponding fields. Record fields are accessed via the $\cdot$ operator (the extraction operator), so if r is a record, then $r.\mathtt{x}$ is a field of this record. The order in which the fields are listed in $\{\overline{\ell_i = e_i}\}$ and $\{\overline{\ell_i : \tau_i}\}$ matters (at least for our small example); that is, the type of the record value `{age=50, name="Matt"}` is `{age:int, name:String}`, which is different from `{name:String, age:int}`, the type of the record value `{name="Matt", age=50}`. Thus, we write the type of a record as

```
{age=50, name="Matt"}:{age:int, name:string}
```

and use the extraction operator to access individual field values, such as

```
{age=50, name="Matt"}.age
```

Typing Rules

In the same way that tuples have a type system, we can do the same for records. Here are the typing rules for the terms dealing with record types:

$$\frac{(\forall i)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{\overline{\ell_i = e_i}\} : \{\overline{\ell_i : \tau_i}\}}(1) \qquad \frac{\Gamma \vdash e : \{\overline{\ell_i : \tau_i}\}}{\Gamma \vdash e.\ell_i : \tau_i}(2)$$

$$\frac{(\forall i) : (\exists j) : \ \ell'_i :=_{\tau} \ell_j \wedge \tau_j \prec_{\tau} \tau'_i}{\{\overline{\ell_j : \tau_j}^{\,j \in \{1..m\}}\} \prec_{\tau} \{\overline{\ell'_i : \tau'_i}^{\,i \in \{1..n\}}\}}(3)$$

Figure 36. Typing rules for a record

Γ contains all of the information needed to type check expressions, as well as the types of elements a record contains. The first rule states that for each expression e of type τ, the form $\{\overline{\ell_i = e_i}\}$ evaluates to a record, which has a record type $\{\overline{\ell_i : \tau_i}\}$ in the typing context Γ. The second rule is the selection operation, which allows us to select a component of a record under the assumption that such component − annotated with a label ℓ − has type τ. Therefore, if an expression e is a record type in the typing context Γ, then we can access each field of record e using the $\cdot$ operator. The third rule is a little bit more involved as it requires to take into account the *depth* and *width* of records. Consider the example below.

Example 9: Is it required for the subtyping relation between two records to have the same number of fields? Before we answer this question, let us define the type `Person` to be a record type `{fname:string, lname:string}`, that contains two fields *fname* and *lname*, both of type string; that is

```
Person = {name:string, lname:string}
```

Let us also define

```
Student = {name:string, lname:string, id:int}
```

as the type of a record with two string fields and one integer field.

It seems reasonable to say that *Student* could be a subtype of *Person* because *Student* contains all of the fields of *Person*, and those fields have the same type as the *Person*'s fields. Also, observe that the subtype contains more elements than the supertype in this example. Of course, this is identical to the connection between a subclass and a superclass, in which the subclass has all of the superclass's components, which it inherits from the superclass, but may also have components that the superclass lacks. Now, given that a variable of a superclass type can hold a reference to an object of subclass type, then any piece of code that uses a value of type *Person* can use a value of type *Student*. Thus, we get

$$Person :=_{\boldsymbol{\tau}} Student$$

and therefore, we define

$$Student \prec_{\boldsymbol{\tau}} Person$$

□

Bearing this in mind, the final rule says that for two general records types r_1 and r_2, each of the form $\{\ell_1 : \tau_1, ..., \ell_n : \tau_n\}$, if $\tau \prec_{\boldsymbol{\tau}} \tau'$, then this implies that anything of type τ can be used in a context expecting something of type τ'. However, if τ has fewer elements ℓ than τ', then there will be values of type τ for which no corresponding reference exists whenever the context requires something of type τ'.

Finally, one thing to note is that we could have defined the depth and width separately (see Figure 37), but we chose to merge them into a single rule represented by the third rule.

$$\frac{(\forall i)\ \tau_i \prec_{\boldsymbol{\tau}} \tau'_i}{\{\overline{\ell_i : \tau_i}^{\,i \in \{1..n\}}\} \prec_{\boldsymbol{\tau}} \{\overline{\ell_i : \tau'_i}^{\,i \in \{1..n\}}\}}\textbf{depth} \qquad \frac{m \leq n}{\{\overline{\ell_i : \tau_i}^{\,i \in \{1..m\}}\} \prec_{\boldsymbol{\tau}} \{\overline{\ell_i : \tau'_i}^{\,i \in \{1..n\}}\}}\textbf{width}$$

Figure 37. Additional typing rules for a record.

Jiapi's Rules

We will now retrace the steps and rewrite the rules using *Jiapi*. Let us start with the first rule.

$$\frac{(\forall i)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{\overline{\ell_i = e_i}\} : \{\overline{\ell_i : \tau_i}\}} (1)$$

We use the notation $recordvalue(id^R)$ to denote the value of a record with name id^R in Γ. The result of $recordvalue$ is a record value type of the form $\{\overline{\ell_i = e_i}\} : \{\overline{\ell_i : \tau_i}\}$. The definition of $recordvalue$ is:

$$\frac{id^R = \{\ell_i = e_i, ..., \ell_n = e_n\} \in \Gamma}{recordvalue(id^R) = \{\overline{\ell_i = e_i}\} : \{\overline{\ell_i : \tau_i}\}}$$

Additionally, $redcordname$ is a function that returns the name of a record type, and so we have:

$$recordname(id^R = \{\ell_1 : \tau_1, ...\ell_n : \tau_n\}) = id^R$$

When everything is combined, we obtain

$$\begin{array}{rcl} rule1(id) & \Leftrightarrow & \forall x \in Dom(\Gamma) : t^R \leftarrow_{\boldsymbol{\tau}} \Gamma(x) \wedge \\ & & Record?(t^R) \wedge recordname(t^R) = id \Rightarrow \\ & & recordvalue(id), \textbf{nil} \end{array}$$

where $Record_?(\tau)$ is a predicate function that evaluates to either *true* or *false* depending on whether or not τ is a record type, and $Dom(\Gamma)$ is the collection of all variables (identifiers) that exists in Γ. Since $Dom(\Gamma)$ is the collection of variables e_i for $i \in 1..m$, then $\Gamma(e_i)$ represents the type τ_i associated with e_i[49].

The second rule is similar, only we must define a function that either returns a record type's field or allow us to access the record type's field if such record exists.

$$\frac{\Gamma \vdash e : \{\overline{\ell_i : \tau_i}\}}{\Gamma \vdash e.\ell_i : \tau_i} (2)$$

We use the notation $recordfield(R)$ to denote a type environment constructed from the fields of record R and the fields of the superclass of R if such exists. (This is needed for the third and final rule.) Note, the fields in R take precedence over the fields in the superclass

[49] $Dom(\Gamma)$ and $\Gamma(e)$ can be implemented as functions.

of R. The definition of *recordfield* is:

$$\frac{id^R = \{\ell_1 : \tau_1, ...\ell_n : \tau_n\} \in \Gamma}{recordfield(id^R) = \{\!\{\ell_1 : \tau_1, ...\ell_n : \tau_n\ \}\!\}}$$

or

$$\frac{\begin{array}{l} id^R = \{\ell_1 : \tau_1, ...\ell_n : \tau_n\} \in \Gamma \\ id^R_s = \{\ell_1 : \tau_1, ...\ell_n : \tau_m\} \in \Gamma \\ id^R \prec_{\boldsymbol{\tau}} id^R_s \end{array}}{recordfield'(id^R) = recordfield(id^R_s) \bullet \{\!\{\ell_1 : \tau_1, ...\ell_n : \tau_n\ \}\!\}}$$

Putting everything together, we obtain

$$\begin{array}{rcl} rule2(id, \ell) & \Leftrightarrow & \forall x \in Dom(\Gamma) : t^R \leftarrow_{\boldsymbol{\tau}} \Gamma(x) \Rightarrow \\ & & Record?(t^R) \wedge recordname(t^R) = id \Rightarrow \\ & & \forall y \in Dom(\Gamma) : \alpha \leftarrow_{\boldsymbol{\tau}} \Gamma(y) \wedge Record?(\alpha) \wedge t^R \prec_{\boldsymbol{\tau}} \alpha \Rightarrow \\ & & recordfield'(id).\ell, recordfield(id).\ell, \textbf{nil} \end{array}$$

For the third and final rule

$$\frac{(\forall i) : (\exists j) : \ \ell'_i :=_{\boldsymbol{\tau}} \ell_j \wedge \tau_j \prec_{\boldsymbol{\tau}} \tau'_i}{\{\overline{\ell_j : \tau_j}^{j \in \{1..m\}}\} \prec_{\boldsymbol{\tau}} \{\overline{\ell'_i : \tau'_i}^{i \in \{1..n\}}\}} (3)$$

we obtain

$$\begin{array}{rcl} rule3(id, id') & \Leftrightarrow & \forall x, y : x \in Dom(\Gamma) \wedge y \in Dom(\Gamma) \Rightarrow \\ & & t^R \leftarrow_{\boldsymbol{\tau}} \Gamma(x) \wedge \alpha \leftarrow_{\boldsymbol{\tau}} \Gamma(y) \ \wedge Record?(t^R) \wedge Record?(\alpha) \wedge \\ & & recordname(t^R) = id \wedge recordname(\alpha) = id' \Rightarrow \\ & & \forall a, b : a \in recordfield(t^R) \wedge b \in recordfield(\alpha) \Rightarrow \\ & & a.\ell :=_{\boldsymbol{\tau}} b.\ell \wedge b.\ell \prec_{\boldsymbol{\tau}} a.\ell \Rightarrow true, false, false \end{array}$$

Chapter 6

Case Studies

6.1 Introduction

In order to evaluate the effectiveness of our meta-language and the general usefulness of our tool for formal usability, we provide two case studies: Espresso from Chapter 4, and a pedagogical language based on high-level design principles referred to as C-Minor [238]. We initially provide separate descriptions of the two case studies and their results. For each study, we first present their type system specification in *Jiapi*. Next, we show a side by side comparison between the code generated by the tool and their implementation using first-order logic and set notation. After having presented both cases studies, we discuss the results and compare the two case studies. Our approach actually reduces the user effort required to construct a type system for general and domain specific languages in comparison to the verification systems we examined in Chapter 7.

6.2 Espresso

As a first case study, we use a subset of Java, which we introduced at the beginning of this thesis and thoroughly examined in Chapter 4. There, we formalized parts of the type system specification using operational functions (such as *type predicates*, *type equality*, *type equivalence*, and *type assignment compatibility*) and using typed λ-calculus. We now use *Jiapi* to specify the same exact type system. We adhere to the specifications presented in Chapter 4 as closely as possible. However, minor deviations from these specifications are

occasionally required to accommodate the features available in Espresso, which we will name in the following text.

6.2.1 Specification of Espresso

We have only discussed theoretical calculi for language design thus far, but the goal is to implement a language that follows these rules in practice. So we begin by presenting the top-level component of Espresso's type system specification in *Jiapi*, which is the `TSType` class. The specification of primitives and some of the constructed types is then introduced, along with code snippets. We note that, in this subsection, we will omit some specific annotations for auxiliary functions and start with the basic structure of the current implementation of Espresso. Bear in mind that this is the code that the tool generates.

We have implemented all the types in a `TypeSystem` package:

```
M1-Air:EspressoNew matt$ ls -l src/TypeSystem/
total 48
-rw-r--r--@ 1 matt staff 2204 Jun 11 16:22 TSArrayType.java
-rwxr--r--@ 1 matt staff 2832 Jun 11 16:30 TSClassType.java
-rw-r--r--@ 1 matt staff 1084 Jun 30 19:58 TSError.java
-rw-r--r--@ 1 matt staff 3802 Jun 11 16:34 TSMethodType.java
-rwxr--r--@ 1 matt staff 735 Jun 10 14:49 TSNullType.java
-rwxr--r--@ 1 matt staff 3922 Jun 11 16:52 TSPrimitiveType.java
-rwxr--r--@ 1 matt staff 3046 Jun 10 14:39 TSType.java
```

Figure 38. `TypeSystem` package.

The actual type checker will be in a file called `TypeChecker.java` in the package `TypeChecker`:

```
package TypeChecker;

import Utilities.Visitor;
import AST.*;
import TypeSystem.*;

public class TypeChecker extends Visitor<Object> {
   ...
}
```

6.2.2 The Semantic Type System

The type system utilizes types classes starting with the prefix *TS*. The super class of all semantic types (which are created from a *clause* declaration) is the *TSType* (implemented in the file `TSType.java`). Naturally, this super class is the place where we want all predicates, functions, and "global" variables to exists. This is because many derived classes can reuse or utilize these fields as they are eventually converted into member fields of the super class in the generated code. The generated class for `TSType` is given below:

```
public abstract class TSType {
  public abstract boolean equal(TSType other);
  public abstract boolean equivalent(TSType other);
  public abstract boolean assignmentCompatible(TSType other);
  public abstract String typeName();
  ...
}
```

As we can see, there are four abstract methods that must be implemented – three of which are directly related to type checking; these are as follows:

- *equal()* – the type operator $=_{\tau}$.
- *equivalent()* – the type operator $\sim_{\tau}$.
- *assignmentCompatible()* – the type operator $:=_{\tau}$.
- and finally *typeName()* which technicality is not needed for the semantic type checking, but is used to produce readable output and user-friendly error messages.

In addition to these four methods, a number of *predicates* will be generated based on the type system. One predicate for each primitive type and constructed type will be generated. For example, from this atomic type specification (the complete specification is given in Figure 39 on page 154)

```
atomic {
 boolean,
 string,
 void,
  [byte < short < int < long < float < double],
  [char < int]
}
```

the following code is generated:

```
...
// integer_?()
public boolean isInteger() {
  return false;
}

// long_?()
public boolean isLong() {
  return false;
}
...
```

As a default, all these are implemented to return *false* in this class. They will be re-implemented to return *true* in the classes that represent the respective types. Finally, the user-defined predicates will also be implemented to return *false*. For Espresso, we have just two:

```
// Numeric_?()
public boolean isNumeric() {
  return false;
}

// Integral_?()
public boolean isIntegral() {
  return false;
}
```

6.2.3 The Primitive Types

In Espresso, not everything is an object. A subset of data types known as primitive types are used quite often. Recall, the list of primitive types in Espresso is as follows:

Table 17. Espresso's primitive types

Name	Numeric	Integral
byte	✓	✓
short	✓	✓
char	✓	✓
integer	✓	✓
long	✓	✓
float	✓	
double	✓	
string		
void		

In the implementation, rather than creating a separate file for each primitive type, we bundle them together into a single primitive type file. A class `TSPrimitive` is implemented as follows:

```
public class TSPrimitive extends TSType {

  // Type lattice of primitive types
  public final static boolean[][] atomicHierarchy = ...

  // Constants defining the primitive types
  public final static int BOOLEAN = ...
  public final static int CHAR = ...
  public final static int BYTE = ...
  public final static int SHORT = ...
  public final static int INT = ...
  public final static int LONG = ...
  public final static int FLOAT = ...
  public final static int DOUBLE = ...
  public final static int STRING = ...
```

```
  public final static int VOID = ...

  private static String[] names = new String[]{
     "boolean", "char", ... };
  ...
}
```

We have associated constructors and accessors as follows:

```
private int kind;

public TSPrimitive(int kind) {
  this.kind = kind;
}

public int getKind() {
  return kind;
}

public String typeName() {
  return names[kind];
}
```

which can be used to create new primitive types as well as access data. We need to re-implement all the predicates that relate to the primitive types:

```
// int_?()
public boolean isInt() {
  return kind == INT;
}

// boolean_?()
public boolean isBoolean() {
  return kind == BOOLEAN;
}

// byte_?()
```

```
public boolean isByte() {
  return kind == BYTE;
}
...
public boolean isPrimitive() {
  return true;
}
```

and the two user-defined predicates $Numeric_?$ and $Integral_?$ defined as follows:

$$Integral_?(t) = byte_?(t) \vee short_?(t) \vee char_?(t) \vee int_?(t) \vee long_?(t)$$

$$Numeric_?(t) = Integral_?(t) \vee float_?(t) \vee double_?(t)$$

they become:

```
// Numeric_?() -- user defined
public boolean isNumeric() {
  return (isFloat() || isDouble() || isIntegral());
}

// Integral_?() -- user defined
public boolean isIntegral() {
  return (isInteger() || isShort() || isByte() ||
          isChar() || isLong());
}
```

We now need to consider the three type operations that must be defined for all types: $=_\tau, \sim_\tau$, and $:=_\tau$:

$$\alpha =_\tau \beta \Leftrightarrow Primitive_?(\alpha) \wedge Primitive_?(\beta) \wedge \alpha = \beta$$

As we have seen, equality ($=_\tau$) between primitive types is easy. We only check that both types are primitives and that their `kind` fields match. Since we implement this in an object-oriented manner, we can thing of $\alpha =_\tau \beta$ as $\alpha.\ =_\tau (\beta)$; that is, we know that α already is a primitive type, so the requirements are reduced to $Primitive_?(\beta) \wedge \alpha = \beta$. For a primitive type, $\alpha = \beta$ always reduce to whether their kinds are the same (`((TSPrimitiveType)other).getKind() == kind`); therefore we obtain:

```
// =_T
public boolean equal(TSType other) {
  return this.isPrimitive() && other.isPrimitive() &&
   ↪  this.kind == ((TSPrimitive) other).kind;
}
```

And since equivalence for primitives is the same as equality, we have:

$$\alpha \sim_{\mathcal{T}} \beta \Leftrightarrow \alpha =_{\mathcal{T}} \beta$$

which is easily implemented as:

```
// ~_T
public boolean equivalent(TSType other) {
  return equal(other);
}
```

For the final of the three type function (assignment compatible) we have the following:

$$byte <_{\mathcal{T}} short <_{\mathcal{T}} <_{\mathcal{T}} int <_{\mathcal{T}} long <_{\mathcal{T}} float <_{\mathcal{T}} double$$

$$chat <_{\mathcal{T}} int$$

this will create the implementation of $<_{\mathcal{T}}$ as follows:

```
public boolean lessThan(TSType other) {
  if (!(other instanceof TSPrimitive)) {
    return false;
  }
  TSPrimitive otherType = (TSPrimitive) other;
  return atomicHierarchy[kind][otherType.getKind()];
}
```

Again, we can use the $<_{\mathcal{T}}$ function to determine compatibility of primitive types. Note that this techniques also work if the parameters are of class type because $:=_{\mathcal{T}}$ is correctly defined for class types as well.

Remark 6 (Type Lattice and $\lceil \alpha, \beta \rceil_{\tau}$): From Section 4.4.4 on page 76, we can argue that a partial order is a complete lattice having a unique largest element $\top$, a unique smallest element $\bot$, and satisfying the condition $\bot < x < \top$ for every x in the set. In *Jiapi*, elements from lattices are stored in a matrix[50] and accessed using the *id space* and *index space* of a type. The *id space* of each element (i.e., each atomic type) in the set is represented by an identifier, which is a positive *long* value. We may use this representation to store larger sets of elements or to use them as keys in maps. The *index space*, in contrast, is represent by indices that correspond to the position of each element in the set of atomic types. Note that since we have a set of dependencies (i.e., some precedence constraints) of the form "α is less than β", we defined a topological ordering of nodes such that for every pair (τ_i, τ_j) in the type lattice, we have $i < j$. □

That leave just one more user-define type-related function to be handled, namely, the $\lceil \alpha, \beta \rceil_{\tau}$ – the ceiling function. This function relates to the type hierarchy we defined above for numeric types:

$$byte <_{\tau} short <_{\tau} int <_{\tau} long <_{\tau} float <_{\tau} double$$

$$char <_{\tau} int$$

Given two numeric types, this function should return the higher of the two:

$$\lceil \alpha, \beta \rceil_{\tau} = \begin{cases} \beta & \text{if } \alpha <_{\tau} \beta \\ \alpha & \text{otherwise} \end{cases}$$

creating another implementation for the same operator $<_{\tau}$ as follows:

```
public TSPrimitive lessThanType(TSType other) {
  if (!(other instanceof TSPrimitive)) {
    return false;
  }
  TSPrimitive otherType = (TSPrimitive) other;
  if (atomicHierarchy[kind][otherType.getKind()]) {
    return new TSPrimitive(kind);
  }
  return this;
```

[50] This is a 2-dimensional array of boolean values represented by `atomicHierarchy`.

```
}
```

Although the ordering operator is defined for numeric types, it could also be defined to work with constructed types (in Section 5.4 on page 136, we gave an example of $<_\tau$ operator for Protocols). The obvious question is: How does the tool determine which version of the ceiling function to use? If the $<_\tau$ operator is used in a boolean expression, the first version is used; otherwise, the second version is used regardless of whether the returned value is assigned to a variable. For example, consider the following implication expression

```
alpha <T beta => ... , ... alpha <T beta
```

that becomes

```
if (alpha.lessThan(beta)) ...
```

This is because the expected value of the **if** statement's conditional expression in the generated code must be *boolean*. However, if the operator appears anywhere where a boolean type is not expected, the expression becomes

```
if (alpha.lessThan(beta)) { ..., alpha.lessThanType(beta) }
```

This brings another interesting question: if $\mathcal{T}(...)$ resolves to a visit, are we *always* making invocations to the appropriate visitor method? The answer is **no**. A type computation is transformed into a ternary operator that checks whether the type of a syntactic node is null. If it is, a visit takes places and during that operation the type of the node is set. This means that if a node already has a type, then $\mathcal{T}(...)$ directly returns the type of this node

Finally, assignment compatibility can now simply be defined for primitive types as

$$\alpha :=_\tau \beta \Leftrightarrow \beta <_\tau \alpha \vee \alpha =_\tau \beta$$

and be implemented as follows

```
public boolean assignmentCompatible(TSType beta) {
  return beta.lessThan(this) || this.equal(beta);
}
```

the atomic type specification is given below

```
atomic {
 boolean,
 string,
 void,
 [byte < short < char < int < long < float < double],
 [int < float < double]
}

...

clause atomic {
  def Integral?(t) = return byte?(t) \/ short?(t) \/ char?(t)
   ↪  \/ int?(t) \/ long?(t)

  def Numeric?(t) = return Integral?(t) \/ float?(t) \/
   ↪  double?(t)

  def alpha =T beta <=> return Primitive?(alpha) /\
   ↪  Primitive?(beta) /\ alpha.kind == (beta as
   ↪  TSPrimitive).kind

  def alpha ~T beta <=> return alpha =T beta

  def alpha :=T beta <=> return beta <T alpha \/ alpha =T
   ↪  beta
}
```

Figure 39. `TSPrimitive` type specification

6.2.4 The Constructed Types

Array Type

Recall, an array is a collection of values accessed by an index set starting at 0 and ending at $n-1$ if the array is sized to hold n elements. Arrays are dynamically allocated but the size is fixed once allocated. Thus, the size of an array type is typically not known at compile time. This means that the only information that can be kept about an array type is its **base type**.

$$Array(Basetype)$$

This can be specified as follows:

$$Array(type : baseType)$$

A two dimensional integer array will look like this:

$$Array(Array(int))$$

in Java such a type can be specified as `int[][]`. As with the primitive types, the three operators $=_\tau, \sim_\tau$, and $:=_\tau$ must be specified:

$$\alpha =_\tau \beta \Leftrightarrow Array_?(\alpha) \wedge Array_?(\beta) \wedge (\alpha.baseType =_\tau \beta.baseType)$$

Two arrays are *equal* if and only if their base types are equal. For $\sim_\tau$ we have:

$$\alpha \sim_\tau \beta \Leftrightarrow \alpha =_\tau \beta$$

Assignment compatibility is a little more complex; note that the following is **not** allowed in Espresso (or Java):

```
int[] i = ...;
double d[];

d = i; // illegal
```

However, the following **is**:

```
class A { ... }
class B extends A { ... }

...
A[] a = null;
B[] b = ...;
a = b
```

That is, we may assign *null* to an array variable, and we may assign an array of *subclass* to an array of *super class*.

$$\alpha :=_{\tau} \beta \Leftrightarrow Array_?(\alpha) \wedge Array_?(\beta) \wedge$$

$$(Null_?(\beta) \vee (Primitive_?(\alpha.baseType) \wedge \alpha.baseType =_{\tau} \beta.baseType) \vee$$

$$(\alpha.baseType :=_{\tau} \beta.baseType))$$

This may look a little complicated, but the implementation is surprisingly simple:

```
public boolean assignmentCompatible(TSType beta) {
  if (this.isArray() && beta.isArray()) {
     TSArray beta2 = (TSArray) beta;
     return (beta.isNull() || (this.baseType.isPrimitive() &&
      ↪  this.baseType.equal(beta2.baseType)) ||
      ↪  (this.baseType.equivalent(beta2.baseType)))
  }
  return false;
}
```

The array type specification is defined as

```
constr Array (baseType:type)

...

clause Array {
  def Array?(t) = return true

  def alpha =T beta <=> {
```

```
    Array?(alpha) /\ Array?(beta) => {
      def beta2 = beta as Array
      return alpha.baseType =T beta2.baseType
    }
    return false;
  }

  def alpha ~T beta <=> return alpha =T beta

  def alpha :=T beta <=> {
    Array?(alpha) /\ Array?(beta) => {
      def beta2 = beta as Array
      return (Null?(beta) \/ (Primitive?(alpha.baseType) /\
       ↪  alpha.baseType =T beta2.baseType) \/
       ↪  (alpha.baseType :=T beta2.baseType))
    }
    return false
  }
}
```

Figure 40. `TSArray` type specification

and its implementation is

```
class TSArray {
  public boolean isArray() {
    return true;
  }
  ...
  public TSType baseType;

  public TSArray(TSType baseType) {
    this.baseType = baseType;
  }
}
```

Null Type

Since Espresso (and Java) have dynamically allocated objects and arrays, a *null* value is needed. It could be implemented as a primitive type value, but here we have a special type, namely, *NullType*.

$$NullType$$

and the three mandatory predicates:

$$\alpha =_{\tau} \beta \Leftrightarrow Null_{?}(\alpha) \wedge Null_{?}(\beta)$$

$$\alpha \sim_{\tau} \beta \Leftrightarrow \alpha =_{\tau} \beta$$

$$\alpha :=_{\tau} \beta \Leftrightarrow false$$

Note that the assignment compatible would require a *null* on the left, which is never allowed. The implementation for *NullType* is therefore

```
public boolean equal(TSType beta) {
  return this.isNull() && beta.isNull();
}

public boolean equivalent(TSType beta) {
  return this.equal(beta);
}

public boolean assignmentCompatible(TSType beta) {
  return false;
}
```

The null type specification is as follows

```
constr Null

...

clause Null {
  def Null?(t) = return true
```

```
  def alpha =T beta <=> return Null?(alpha) /\ Null?(beta)

  def alpha ~T beta <=> return alpha =T beta

  def alpha :=T beta <=> return false
}
```

Figure 41. `TSNull` type specification

and its implementation is defined as

```
public class TSNull {
  public boolean isNull() {
    return true;
  }
  ...

  public TSNull() {}
}
```

Class Type

Apart from arrays, classes are the only other constructed type in Espresso (Java has enumerations as well, Espresso does not). Since classes create a class hierarchy and since a variable of super class can hold a reference to a subclass, a class type must contain information about its super classes and interfaces:

$$Class(Name, SuperClass^{?}, Interface^{*})$$

A class type uses that class' name, and the super class and the interfaces are class types as well.

$$Class(name : id, superClass^{?} : Class, interfaces^{*} : Class)$$

$$\alpha =_{\tau} \beta \Leftrightarrow Class_{?}(\alpha) \wedge Class_{?}(\beta) \wedge \alpha.name = \beta.name$$

$$\alpha \sim_{\tau} \beta \Leftrightarrow \alpha =_{\tau} \beta$$

$$\alpha :=_{\tau} \beta \Leftrightarrow Class_?(\alpha) \wedge Class_?(\beta) \wedge (Null_?(\beta) \vee \alpha \leq_{\tau} \beta)$$

From the above formulas, we can see that $:=_{\tau}$ uses a different operator, namely, $\leq_{\tau}$. In object-oriented terms, $\leq_{\tau}$, is often written as <:. That is the "is super class" operator. If class A extends class B, then $A <: B$ and $A \leq_{\tau} B$. Remember, the <: operator is represented by $\prec_{\tau}$ with the arrow pointing to the class that extends the super class (i.e., $B \prec_{\tau} A$).

$$\alpha \leq_{\tau} \beta \Leftrightarrow \alpha =_{\tau} \beta \ \vee (\alpha.superClass \neq null \wedge \alpha \leq_{\tau} \beta.superClass) \ \vee (\bigvee_{\delta \in \beta.interfaces} \alpha \leq_{\tau} \delta)$$

The implementation generated from the above formulas is

```
public boolean equal(TSType beta) {
  if (this.isClass() && beta.isClass()) {
    TSClass beta2 = (TSClass) beta;
    return this.isClass() && this.name.equals(beta2.name);
  }
  return false;
}

public boolean equivalent(TSType beta) {
  return this.equal(beta);
}

public boolean assignmentCompatible(TSType beta) {
  return this.isClass() && beta.isClass() && (beta.isNull() ||
  ↪  this.lessThan(beta));
}

public boolean lessThan(TSType beta) {
  TSClass beta2 = (TSClass) beta;
  boolean binVar0 = false;
  for (int delta = 0; delta < beta2.interfaces.size();
  ↪  ++delta) {
    binVar0 |= this.lessThan(beta2.interfaces[delta]);
  }
```

```
  return this.equal(beta2) || (this.superClass != null &&
   ↪  this.lessThan(beta2.superClass)) || binVar0;
}
```

and its specification is as follows

```
constr Class (name:id, superClass?:Class, interfaces*:Class)

...

clause Class {
  def Class?(t) = return true

  def alpha =T beta <=> {
    Class?(alpha) /\ Class?(beta) => {
      def beta2 = beta as Class
      return Class?(alpha) /\ alpha.name == beta2.name
    }
    return false
  }

  def alpha ~T beta <=> return alpha =T beta

  def alpha :=T beta <=> return Class?(alpha) /\ Class?(beta)
   ↪  /\ (Null?(beta) \/ alpha <=T beta)

  def alpha <=T beta <=> {
    Class?(alpha) /\ Class?(beta) => {
      def beta2 = beta as Class
      return alpha =T beta2 \/ (alpha.superClass != nil /\
       ↪  alpha <=T beta2.superClass) \/  (\/ delta in
       ↪  0..|beta2.interfaces|: alpha <=T
       ↪  beta2.interfaces[delta])
    }
    return false
  }

  {:
```

```
    public ClassDecl myDecl;
    public Class(ClassDecl classDecl) {
      myDecl = classDecl;
      name = myDecl.name();
    }
   :}
}
```

Figure 42. `TSClass` type specification

Method Type

Finally, we need method types. Even though Espresso does not allow passing methods as parameters, it makes sense to have a method type for specifying the way a target method is found in an invocation.

A method type depends on the name of the method, the type of the parameters and the return type:

$$Method(Name, ParamType^*, ReturnType)$$

or in the type system language:

$$Method(name : id, paramTypes^* : type, returnType : type)$$

$$\alpha =_{\boldsymbol{\tau}} \beta \Leftrightarrow Method_?(\alpha) \wedge Method_?(\beta) \wedge (\alpha.name = \beta.name) \wedge$$

$$(|\ \alpha.paramTypes\ |=|\ \beta.paramTypes\ |) \wedge$$

$$\left(\bigwedge_{i \in \{0..|\alpha.paramTypes|-1\}} (\alpha.paramTypes)_i =_{\boldsymbol{\tau}} (\beta.paramTypes)_i \right)$$

$$\alpha \sim_{\boldsymbol{\tau}} \beta \Leftrightarrow \alpha :=_{\boldsymbol{\tau}} \beta \wedge \beta :=_{\boldsymbol{\tau}} \alpha$$

It should be pointed out that we use equivalence in terms of assignment compatibility in both directions; we do this as we do not want to compare names like we do in $=_{\boldsymbol{\tau}}$.

$$\alpha :=_{\boldsymbol{\tau}} \beta \Leftrightarrow Method_?(\alpha) \wedge Method_?(\beta) \wedge (|\ \alpha.paramTypes\ |=|\ \beta.paramTypes\ |) \wedge$$

$$\left(\bigwedge_{i \in \{0..|\alpha.paramTypes|-1\}} (\alpha.paramTypes)_i :=_\tau (\beta.paramTypes)_i \right)$$

If we planned to allow the passing of methods as parameters, we would also have to do some checking of the return type, but that is not necessary here. Also, note that we do not check for the equality of names. The implementation for method types is as follows

```
public boolean equal(TSType beta) {
  if (this.isMethod() && beta.isMethod()) {
    TSMethod beta2 = (TSMethod) beta;
    boolean binVar1 = true;
    for (int i = 0; i < this.paramTypes.size() - 1; ++i) {
     binVar1 &=
      ↪  (this.paramTypes)[i].equal((beta2.paramTypes)[i]);
    }
    return (this.name.equals(beta2.name)) &&
     ↪  (this.paramTypes.size() == beta2.paramTypes.size()) &&
     ↪  binVar1;
  }
}

public boolean equivalent(TSType beta) {
  return this.assignmentCompatible(beta) &&
   ↪  beta.assignmentCompatible(this);
}

public boolean assignmentCompatible(TSType beta) {
  if (this.isMethod() && beta.isMethod()) {
    TSMethod beta2 = (TSMethod) beta;
    boolean binVar2 = true;
    for (int i = 0; i < this.paramTypes.size() - 1; ++i) {
      binVar2 &=
       (this.paramTypes)[i].equivalent((beta2.paramTypes)[i]);
    }
    return (this.paramTypes.size() == beta2.paramTypes.size())
     ↪  && binVar2;
  }
}
```

and its specification is given bellow

```
constr Method (name:id, paramTypes*:type, returnType:type)

...

clause Method {
  def Method?(t) = return true

  def alpha =T beta <=>  {
    Method?(alpha) /\ Method?(beta) => {
      def beta2 = beta as Method
      return (alpha.name == beta2.name) /\
      (|alpha.paramTypes| == |beta2.paramTypes|) /\
      (/\ i in 0..|alpha.paramTypes| - 1 :
       ↪  (alpha.paramTypes)[i] =T (beta2.paramTypes)[i])
    }
    return false
  }

  def alpha ~T beta <=> return alpha :=T beta /\ beta :=T
   ↪  alpha

  def alpha :=T beta <=> {
    Method?(alpha) /\ Method?(beta) => {
      def beta2 = beta as Method
      return (|alpha.paramTypes| == |beta2.paramTypes|) /\
      (/\ i in 0..|alpha.paramTypes| - 1 :
       ↪  (alpha.paramTypes)[i] :=T (beta2.paramTypes)[i])
    }
    return false
  }

  {: public ClassDecl myDecl; :}
}
```

Figure 43. `TSMethod` type specification

6.3 From `Parse Tree` to *Types*

In any language we want to type check, there are certain parts of the parse tree that represent a type in the language. A simple example is the primitive types like `int` and `double`. These parse-tree representations of types must be turned into actual types in the type system, that is:

$$int_{\text{parse tree}} \rightsquigarrow int_{\text{type system}}$$

or, if we use `teletype font` for parse tree representations and *italics* for type system representations:

$$\texttt{int} \rightsquigarrow int$$

For example, the *visit()* method for turning `int` into *int* could look like this:

```
public TSType visitPrimitiveType(PrimitiveType pt) {
  // Create a new Type System
  TSPrimitive tspt = new TSPrimitive(pt.getKind());
  pt.setTStype(tspt);
  return tspt;
}
```

It is worth noting, the integer constants used for the `parse tree` primitive types are also used by the *Type System* primitive type. This does not have to be the case, but it does make things easier. Moreover, we have added some "back-hooks" in the parse tree nodes that hold values of types in the *Type System*. More specifically, `Type.java`, `ClassBodyDecl` and `VarDecl` all have the following code:

```
// Type System Related Stuff
private TSType myType;

public void setTStype(TSType type) {
  myType = type;
}
```

```
public TSType TStype() {
  return myType;
}
// End of Type System Related Stuff
```

6.3.1 Constructed Types

We begin this section by revisiting the constructed types we just discussed. We can now go through some of the parse three nodes that require type checking and apply the techniques that we learned. Then, in a re-implementation of the visitor pattern, we can implement these techniques as methods. Let us start with an example of *locals* and *parameter* declarations.

Locals and Parameter Declarations

Both local and parameter declarations are of the form:

$$type\ name\ [\ =\ expression\ ]$$

by visiting the type we obtain a *Type System* type and we can set the type of the local or parameter. Here is an example of how to do this for a *ParamDecl*:

```
public TSType visitParamDecl(ParamDecl pd) {
  TSType tsType = (TSType)pd.type().visit(this);
  pd.type().setTStype(tsType);
  return tsType;
}
```

and here is its description

ParamDecl

Syntactic Node:
ParamDecl: pd

Accessors:

$Type\ type = pd.type()$

Rules:
$\mathcal{T}(pd) \leftarrow_{\mathcal{T}} \mathcal{T}(type)$

Checks:
• None

followed by its specification written in *Jiapi*

```
action ParamDecl for pd {
  type: Type <- {: pd.type() :}
  T(pd) <- T(type)
  return T(pd)
}
```

Let us briefly consider what type checking is needed for the parse three nodes, but first it is worth mentioning the following:

1. When the parser creates a node in the parse tree for a literal value, it assigns a type to it.

2. Statements do not have values; if an error is detected within a statement, the type error (i.e., *ErrorType*) is assigned.

3. Error types do not resolve to anything by themselves.

4. A named type is resolved to an *actual type*.

Arrays

To indicate that a type is an array type, it gets wrapped in an *ArrayType* for each array level that it has. For example, `int[][]` is an *ArrayType* with type *int* and dimension 2, becoming $ArrayType(ArrayType(int))$

ArrayType

Syntactic Node:
ArrayType: at

Accessors:
$Type\ baseType = at.baseType()$
$int\ depth = at.getDept()$

Rules:
$\mathcal{T}(at) \leftarrow_{\mathcal{T}} Array^{depth}(\mathcal{T}(baseType))$

Checks:
- Add the correct number of *Arrays*
- Set *myDecl* to point to the top array

We can implement an *ArrayType* as follows:

```
action ArrayType for at {
  def baseType: Type <- {: at.baseType :}
  def depth: int <- {: at.getDepth() :}
  def bastType <- T(baseType) as TSType
  def at_t <- baseType
  forall i in 0..depth : at_t <- new TSArray(at_t)
  T(at) <- T(at_t)
  return T(at)
}
```

Note, we add the correct numbers of *ArrayType*(...)s and find the element type, meaning, the type without *ArrayType* round it.

Type checking *ArrayLiteral*s is not quite simple. We cannot determine the definite type of an array literal, but we can determine if an array literal can be assigned to an array type by writing a fairly simple recursive procedure *arrayLiteralAssignmentCompatible*, which takes in an array type and an array literal. Recall, from Section 4.4.5 on page 83 we know that any array type can be assigned the value { }, and when we get to non-array items, the value

must be assignment compatible with the array type's base type. This is depicted in Table 19.

Table 19. Example of *arrayLiteralAssignmentCompatible*.

Level	Literal	Type	
	{{{30, 40}, {}}, {{}, {50, 60}}}	**double**[][][]	
[0]	{{30, 40}, {}}	**double**[][]	
[1]	{{}, {50, 60}}	**double**[][]	
[2]	{}	**double**[][]	✓
[0][0]	{30, 40}	**double**[]	
[0][1]	{}	**double**	✓
[0][1][0]	30	**double**	✓
[0][1][1]	40	**double**	✓
[1][0]	{}	**double**[]	✓
[1][1]	{50, 60}	**double**[]	
[1][1][0]	50	**double**	✓
[1][1][1]	60	**double**	✓

As a result, there is no need to visit an *ArrayLiteral* in the type checker; instead, we must perform the checking described above in the *NewArray* (after all, this is the only place where an *ArrayLiteral* can appear). An allocation for an array using **new** is of the form:

$$variable\ name = \textbf{new}\ \{[expression\ list]\};$$

NewArray

Syntactic Node:

NewArray: na

Accessors:

$Type\ baseType = ne.baseType()$

$Sequence\ dimsExpr = ne.dimsExpr()$

$Sequence\ dims = ne.dims()$

$ArrayLiteral\ init = ne.init()$

Rules:

- $(\forall\, dim \in dimsExpr : \neg Integral_?(\mathcal{T}(dimsExpr_i))) \Rightarrow$

$Error$("Array dimension must be of integral type.")

- $(init \neq null \wedge \neg(ALAC(\mathcal{T}(ne), \mathcal{T}(init)))) \Rightarrow$
  $Error$("Array Initializer is not compatible.")
- $\mathcal{T}(at) \leftarrow_{\mathcal{T}} Array^{depth}(\mathcal{T}(baseType))$

Checks:

- The type of *dim* must be integer if it is present
- If there is an *initializer* (i.e.,$\{e_1, \ldots, e_n\}$), we need to check if it is of proper and equal depth, and then visit every element in it and check if they are assignment compatible with the base type.

The first thing we check for a *NewArray* expression is that all of the non-empty dimensions are of integral type, which means that something like this cannot exist: `new int[3.4]`. If there is an initializer (in which case all dimensions will be empty), we simply call the *arrayLiteralAssignmentCompatible* method that we previously described.

The implementation for a *NewArray* is as follows:

```
action NewArray for ne {
  def baseType: Type <- {: ne.baseType :}
  def dimsExpr: Sequence <- {: ne.dimsExpr() :}
  def dims: Sequence <- {: ne.dims() :}
  def init: ArrayLiteral <- {: ne.init() :}
  forall dims in 0..|{: dimsExpr.size() :}|:
    !Integral?(T(dimsExpr[i])) =>
       Error("Array dimension must be of integral type.")
  def array_t <- T(basetype)
  forall i in 0..|{:dims.nchildren + dimsExpr.nchildren:}| :
     array_t <- new TSArray(array_T, null)
  T(ne) <- array_t
  init != null /\ !(ALAC(T(ne), T(init))) =>
     Error("Array initializer is not compatible.")
  return array_t
}
```

An *ArrayAccessExpr* is made up of an expression and one index in `[]` (each set of brackets has one parse tree node), so all we need to check here is that the expression is of array type.

Naturally, the type of the entire parse tree node must be set, which is the base type of the array type if the type is only one dimension deep; otherwise, it is a new array type with the same base type as the expression with one dimension less. The form of an *ArrayAccessExpr* is:

$$expr_1[expr_2]$$

ArrayAccessExpr

Syntactic Node:
ArrayAccessExpr: ae

Accessors:
$Expression\ target = ae.target()$
$Expression\ index = ae.index()$

Rules:
- $(Error_?(\mathcal{T}(target))) \Rightarrow$
  $\mathcal{T}(ae) \leftarrow_{\mathcal{T}} \mathcal{T}(target)$
- $(Array_?(\mathcal{T}(target))) \Rightarrow \{$
  - $\mathcal{T}(target) \leftarrow_{\mathcal{T}} \mathcal{T}(target).baseType()$
  - $(\neg Integral_?(\mathcal{T}(index)) \wedge \neg Error_?(index)) \Rightarrow$
    $Error(\text{"Array access index must be of integral type"})$

  $| \ \mathcal{T}(ae) \leftarrow_{\mathcal{T}} Error$

$\}$

Checks:
- The type of $expr_1$ must be *ArrayType*, and the type of $expr_2$ must be integer

The implementation for *ArrayAccessExpr* is:

```
action ArrayAccessExpr for ae {
  def target: Expression <- {: ae.target() :}
  def index: Expression <- {: ae.index() :}
  Error?(T(target)) => T(ae) <- T(target)
    , Array?(T(target)) => {
```

```
        T(ae) <- T(ae).baseType()
        !Integral?(T(index)) /\ !Error?(T(index)) =>
            Error("Array access must be of integral type")
      } , T(ae) <- T(target)
  return T(ae)
}
```

Class

Finally, a simple class declaration is of the form:

[*modifier*] **class** *class name*
[**extends** *class name* [**,** *class name*]]
// fields, constructors, static initializer are defined here...

ClassDecl

Syntactic Node:
ClassDecl: cd

Accessors:
$ClassType\ superClass = cd.superClass()$
$Sequence\ interfaces = cd.interfaces()$
$Sequence\ body = cd.body$

Rules:
- $\mathcal{T}(superClass)$
- $\mathcal{T}(interfaces)$
- $\mathcal{T}(body)$

Checks:
- Create a *ClassType* and use it to set the type of the class
- Update the current class as needed when traversing the type system

As we can see, class types can be simple or instances of more complex class declarations (as described in Chapter 4). Again, unlike Java, Espresso does not allow the declaration of inner classes or generic classes, making type checking easier in comparison to Java's. At this

point, one may wonder about *class types* (after all a class is a type). We could also write the specification for a class type, which would only involve creating the type for the current class and then setting it to point to its declaration (wherever that may be in the source file). This approach would, of course, be similar to what we did with *ParamDecl*. The implementation for a *ClassDecl* is therefore

```
action ClassDecl for cd {
  def superClass: ClassType <- {: cd.superClass() :}
  def interfaces: Sequence <- {: cd.interfaces() :}
  def body: Sequence <- {: cd.body() :}
  T(superClass)
  T(interfaces)
  T(body)
  return T(cd)
}
```

Assignment

As a final example, consider assignments in Espresso. Generally speaking, an assignment is a statement that adds a new value to a program entity, which is indicated by the left-hand side. Also, expressions must be used on both the left and right sides of an assignment. The form of an assignment in Espresso is

$$target = value$$

Type checking code for assignments is not as straightforward as we might think due to the different cases that we must take into account. Consider an assignment of the form $id = e$, where id is an identifier. When our id lookup results in the declaration of a local variable (which could be a formal parameter), a compile-time error occurs if the static type of e is not assignable to the declared type of the local variable. Additionally, further analysis as well as evaluation of $p.id = e$ is needed if the declaration happens to be a field, where p is an important prefix and id is (again) an identifier. For example, what if id is a non-static field? A compile-time error should occur, unless p has a member such that the static type of e is assignable to the declared type.

Assignment

Syntactic Node:

Assignment: as

Accessors:

$Expression\ left = as.left()$

$Expression\ right = as.right()$

$AssignmentOp\ op = as.op()$

Rules:

- $Error_?(\mathcal{T}(right)) \Rightarrow$

  ϵ

  | $\{:$ `op.kind == AssignmentOp.EQ` $:\} \Rightarrow$

  $\neg(\mathcal{T}(left) :=_{\mathcal{T}} \mathcal{T}(right)) \Rightarrow$

  $Error(...)$

  | $(\{:$ `op.kind == AssignmentOp.MULTEQ` $:\} \vee$

  $\{:$ `op.kind == AssignmentOp.DIVEQ` $:\})$

  $(\{:$ `op.kind == AssignmentOp.MODEQ` $:\} \vee$

  $\{:$ `op.kind == AssignmentOp.PLUSEQ` $:\})$

  $(\{:$ `op.kind == AssignmentOp.MINUSEQ` $:\}) \Rightarrow$

  $((\{:$ `op.kind == AssignmentOp.PLUSEQ` $:\}) \wedge string_?(\mathcal{T}(left))) \Rightarrow$

  ϵ

  | $(\neg(\mathcal{T}(left) :=_{\mathcal{T}} \mathcal{T}(right))) \Rightarrow$

  $Error(...)$

  | $(\neg(Numeric_?(\mathcal{T}(left) \wedge Numeric_?(\mathcal{T}(right)))) \Rightarrow$

  $Error(...)$

  | $(\{:$ `op.kind == AssignmentOp.LSHIFTEQ` $:\} \vee$

  $\{:$ `op.kind == AssignmentOp.RSHIFTEQ` $:\})$

  $(\{:$ `op.kind == AssignmentOp.RRSHIFTEQ` $:\}) \Rightarrow$

  $(\neg(Integral_?(\mathcal{T}(left)) \wedge Integral_?(\mathcal{T}(right)))) \Rightarrow$

  $Error(...)$

  | $(\{:$ `op.kind == AssignmentOp.ANDEQ` $:\} \vee$

  $\{:$ `op.kind == AssignmentOp.OREQ` $:\})$

  $(\{:$ `op.kind == AssignmentOp.XOREQ` $:\}) \Rightarrow$

  $(\neg(((Integral_?(\mathcal{T}(left))) \wedge Integral_?(\mathcal{T}(right))) \vee$

$$((boolean_?(\mathcal{T}(left))) \wedge boolean_?(\mathcal{T}(right)))) \Rightarrow$$
$$Error(...)$$
$$\mid \epsilon$$
- $T(as) \leftarrow_{\mathcal{T}} \mathcal{T}(left)$

Checks:

- The type of *target* must be assignment compatible with the with the type of *value*

Even though there are a few other checks that we did not include here, the implementation of an *Assignment* is given as follows

```
action Assignment for as {
  def left: Expression <- {: as.left() :}
  def right: Expression <- {: as.right() :}
  def op: AssignmentOp <- {: as.op() :}
  def left_t <- T(left)
  Error?(left_t) => ;
  , {: op.kind == AssignmentOp.EQ :} =>
    !(T(left) :=T T(right)) => Error("...")
  , {: op.kind == AssignmentOp.MULTEQ :} \/ {: op.kind ==
   ↪  AssignmentOp.DIVEQ :} \/
    {: op.kind == AssignmentOp.MODEQ: } \/ {: op.kind ==
     ↪  AssignmentOp.PLUSEQ :} \/
    {: op.kind == AssignmentOp.MINUSEQ :} =>
       {: op.kind == AssignmentOp.PLUSEQ :} /\
        ↪  string?(T(left)) =>
          ;
       , !(T(left) :=T T(right)) => Error("...")
       , !(Numeric?(T(left)) /\ Numeric?(T(right))) =>
            Error("...")
  , {: op.kind == AssignmentOp.LSHIFTEQ :} \/ {: op.kind ==
   ↪  AssignmentOp.RSHIFTEQ :} \/
    {: op.kind == AssignmentOp.RRSHIFTEQ:} =>
       !(Integral?(T(left)) /\ Integral?(T(right))) =>
           Error("...")
  , {: op.kind == AssignmentOp.ANDEQ :} \/ {: op.kind ==
   ↪  AssignmentOp.OREQ :} \/
```

```
    {: op.kind == AssignmentOp.XOREQ :} =>
       !(Integral?(T(left)) /\ Integral?(T(right)) \/
         boolean?(T(left)) /\ boolean?(T(right))) =>
           Error("...")
  , ;
  T(as) <- T(left)
  return T(as)
}
```

6.4 C-Minor

We add a second, distinct case study to strengthen the results of this thesis and to demonstrate that our approach can be applied to several type system specifications. Our second case study is a pedagogical language that has been proposed at the University of Nevada Las Vegas (UNLV) to facilitate the learning of programming languages for first years college students.

6.4.1 Introduction

Although C-Minor, as its name suggests, follows largely C-style syntax, it borrows elements from many other languages such as Fortran 90, Ada, LISP, Java, and Pascal. The main paradigms of the language are imperative and object-oriented. It is a language designed specifically for novice programmers, focusing on features that are useful to users at that level while avoiding overly complex constructs that they do not require [238].

6.4.2 Types

Every value in C-Minor has a data type that instruct the interpreter what type of data is being specified so that it knows how to work with that data [238]. Boolean, integer, character, and enumeration types are among the discrete types (as named in type declarations). Keep in mind that while C-Minor is *statically typed* language, which means that it must know the types of all variables, it support for range values for data types. Therefore, the programmer

can specify a range for each integer variable, and the compiler will choose the appropriate representation. Let us write a specification for the atomic integer type.

To begin, an integer type in C-Minor must be represented as a constructed type rather than as an atomic. This is done to avoid having the same property (the ability to specify a range of values) for all of the other atomic types. Of course, this raises a problem because we have to represent an atomic type as a constructed type. So, is it still possible to construct a type lattice for the set of atomic types in C-Minor? The answer is *yes*; however, in order for the operational functions to work (i.e., **type predicates**, **type equality**, **type equivalence**, **type assignment compatibility**), the remaining atomic types must also be interpreted as constructed types.

We can define an integer type as follows

$$Atomic(name, [low..high]^{?})$$

with an optional range, where *low* and *high* can be used to specify the default range of acceptable values for each atomic type, while *name* is used for name equality (mainly to enforce type checking).

Recall, we need to consider the three operators: $=_{\boldsymbol{\tau}}, \sim_{\boldsymbol{\tau}}, \text{and } :=_{\boldsymbol{\tau}}$; therefore, for an atomic type, we say that two types alpha and beta are equal if their names are the same and their range of values are equal or if one is within the range of the other.

$$\alpha =_{\boldsymbol{\tau}} \beta \Leftrightarrow Atomic_{?}(\alpha) \wedge Atomic_{?}(\beta) \wedge ((\alpha.low \leq \beta.low \wedge \alpha.high \leq \beta.high) \vee$$

$$(\beta.low \leq \alpha.low \wedge \beta.high \leq \alpha.high)) \wedge (\alpha.name = \beta.name)$$

For type equivalence we have

$$\alpha \sim_{\boldsymbol{\tau}} \beta \Leftrightarrow \alpha =_{\boldsymbol{\tau}} \beta$$

and for assignment compatibility we have

$$\alpha :=_{\boldsymbol{\tau}} \beta \Leftrightarrow \beta <_{\boldsymbol{\tau}} \alpha \wedge \alpha =_{\boldsymbol{\tau}} \beta$$

The implementation from the above formulas is as follows

```
public boolean equal(TSType other) {
  if (this.isAtomic() && other.isAtomic()) {
    return this.name.equals(other.name) &&
    ((this.low <= other.low && this.high <= other.high) ||
    (other.low <= this.low && other.high <= this.high)) &&
    (this.name.equals(other.name))
  }
  return false;
}

public boolean equivalen(TSType other) {
  return equal(other);
}

public boolean assignmentCompatible(TSType beta) {
  return beta.lessThan(this) || this.equal(beta);
}
```

6.4.3 Other Similar Constructs

Interestingly, C-Minor features a relatively free form looping structure. The block associated with the **loop** statement repeats until the **until** statement's boolean expression evaluates to true. That is, the loop's conditional is an ending condition rather than a continuation condition, and it can be placed anywhere in the loop's sequence. No "break" style command exists to terminate the loop outside of the **until statement**. To simulate a C/C++ or Java style while loop, the **until** should be the first statement in the block. A do-while loop would have its **until** condition as the last statement in the block. A for-loop would have a limited implementation as an iterator.

In order for the type checker to work, we will assume that C-Minor's loop structure generates three types of loop statements: **while**, **do-while**, and **for** as abstract syntax trees, which can then be traversed based on where the until condition appears. The description for a while statement transformation from a **until** loop is given below

WhileStat

Syntactic Node:
WhileStat: ws

Accessors:
$Expression\ expr = ws.expr()$
$Statement\ stat = ws.stat()$

Rules:
- $(Error_?(\mathcal{T}(expr))) \Rightarrow$
  $\mathcal{T}(ws) \leftarrow_{\mathcal{T}} void$
  $\mid\ (Boolean_?(\mathcal{T}(expr))) \Rightarrow$
  $\mathcal{T}(ws) \leftarrow_{\mathcal{T}} void$
  $\mid\ \mathcal{T}(ws) \leftarrow_{\mathcal{T}} Error(\text{"Expression in while statement must be of Boolean type."})$
- $\mathcal{T}(stat)$

Checks:
- $Boolean_?(\mathcal{T}(expr))$

The implementation of this while statement is given as follows

```
action WhileStat for ws {
  def expr: Expression <- {: ws.expr() :}
  def stat: Statement <- {: ws.stat() :}
  def op: BinOp <- {: be.op() :}
  Error?(T(expr)) =>
     T(ws) <- void
   , Boolean?(T(expr)) =>
       T(ws) <- void
     , T(ws) <- Error
  T(stat)
  return T(ws)
}
```

Similarly, we could write a description for a do-while statement for when the **until**'s

conditional is used last in the body of the loop. Below is its description followed by its implementation.

DoStat

Syntactic Node:
DoStat: ds

Accessors:
$Statement\ stat = ds.stat()$
$Expression\ expr = ds.expr()$

Rules:
- $\mathcal{T}(stat)$
- $(Error_?(\mathcal{T}(expr))) \Rightarrow$
  $\mathcal{T}(ds) \leftarrow_{\mathcal{T}} void$
  | $(Boolean_?(\mathcal{T}(expr))) \Rightarrow$
  $\mathcal{T}(ds \leftarrow_{\mathcal{T}} void$
  | $\mathcal{T}(ds) \leftarrow_{\mathcal{T}} Error($"Expression in while statement must be of Boolean type.")

Checks:
- $Boolean_?(\mathcal{T}(expr))$

Notice that since C-Minor shares some similar constructs with Espresso, we can use independent pieces of a previous formalization of a type system, such that parts shared between languages can be reused.

```
action DoStat for ds {
  def expr: Expression <- {: ds.expr() :}
  def stat: Statement <- {: ds() :}
  T(stat)
  op: BinOp <- {: be.op() :}
  Error?(T(expr)) =>
     T(ds) <- void
   , Boolean?(T(expr)) =>
       T(ds) <- void
```

```
      , T(ds) <- Error
  return T(ds)
}
```

6.5 Finding the Correct Method

Let us briefly demonstrate how to compare and identify the right method in both **Espresso** and **C-Minor**. Consider the following piece of code written in **Espresso** [5]:

```
void foo(double d, long l) { ... } ❶
void foo(int i, long l) { ... } ❷
void foo(int i) { ... } ❸
void foo(string s, double d) { ... } ❹
...
void main() {
  int i;
  ...
  foo(i, i + 1);
}
```

Which of the four `foo` methods should be the one called in `main`? The answer is ❷, of course, but why? Let us look at each method one at a time. ❸ is not the correct method call because it has only one parameter. Since ❹'s formal parameter is a *string*, it is also a bad choice. We are left with options ❶ and ❷. If each were the only declaration of `foo`, both would be called,but we must choose **only** one. One way to do this is by specifying a type for each that includes the types of the parameters and they value that each function returns. For example:

$$\mathcal{T}(\texttt{void foo(double d, long l)\{...\}}) = (double, long) \rightarrow void$$

We could also specify that for an expression to be compatible with the method `foo`, it must be of the form $f(expr_1, ..., expr_2)$, where f is `foo` and the number of actual parameters is the number for formal parameters where each has to be assignment-compatible with the equivalent formal parameter; that is, $f(e_1, ..., e_n)$ is compatible with $t\ g(t_1\ p_1, ..., t_m\ p_m)$

which has type $(t_1, ..., t_m) \rightarrow t$ if [5]:

- $f = g$ (the methods are called the same) and
- $n = m$ (they have the same number of parameters) and
- $\forall i(1 \leq i \leq n) : t_i :=_{\mathcal{T}} \mathcal{T}(e_1)$ (using the assignment compatibility rule, each formal parameter is able to hold the value of the actual parameter).

Before implementing an algorithm for the type checker, we will use Definition 11 in order to determine the correct method call by first finding all possible methods, and then removing the more generals one.

Definition 11 (Comparing Methods): Let $T_1 = (t_{1,1}, ..., t_{1,n}) \rightarrow t_1$ and $T_2 = (t_{2,1}, ..., t_{2,m}) \rightarrow t_2$ be two function types. We say that the function with type T_2 is **more general** than the one with type T_1 if:

$$T_1 \leq_{\mathcal{T}} T_2 \Leftrightarrow (n = m) \wedge (\bigwedge_{i=1}^{n} t_{2,i} :=_{\mathcal{T}} t_{1,i})$$

Alternatively, consider a set of *actual parameters* of type $t_{1,1}, ..., t_{1,n}$, and then consider if these could be passed to the function with *formal parameters* $t_{2,1}, ..., t_{2,m}$. If they can, then the method with T_2 is more general than the one with type T_1. Note: this definition **cannot** be used for defining assignment compatibility for function types. □

We shall now implement a method for finding the correct method call, called *findMethod*, which we will refer to as *findMethod*. This method takes in a sequence of concrete methods, the name of the method we are looking to invoke, and a sequence of the actual parameters. Note that since the actual parameters are expressions, and the formal parameters are declarations, we can call $\mathcal{T}(...)$ on both of them. For completeness, we also need to write the specification of an *invocation* in Espresso in order to demonstrate the use of the *findMethod* method.

The resolution of method invocations falls under the purview of the type checker (at least in the case of statically typed languages), so an invocation in Espresso looks like this

```
action Invocation for inv {
  def target:Expression <- {: inv.target() :}
  def params:Sequence <- {: inv.params() :}
  def methodName:Name <- {: inv.methodName() :}

  def target_t:TSType <- nil
  target != nil => target_t <- T(target) , target_t <- new
   ↪  Class({: inv.myClass :})

  !Class?(target_t) => {: System.out.println("Error: Attempt
   ↪  to invoked something not of class type") :}

  def invocation_t:Method <- nil
  def params_t:List[TSType] <- List()
  forall i in 0..|{: params.nchildren :}| : params_t += {:
   ↪  params.children[i].visit(this) :} as TSType

  invocation_t <- new Method({: methodName.getname() :},
   ↪  params_t)

  def candidates:List[Method] <- findMethod((target_t as
   ↪  Class).{: myDecl.allMethods :}, invocation_t, true)

  |candidates| > 1 => {
    {: System.out.println("More than one candidate remains:
     ↪  "); System.exit(0) :}
  }, |candidates| == 0 => {
    {: System.out.println("No candidates found!");
     ↪  System.exit(0) :}
  }, {
    def method:Method <- candidates[0]
    T(inv) <- method.returnType
    {: inv.targetMethod :} <- << method.myDecl >> as
     ↪  MethodDecl
  }
}
```

which generates the following implementation of an invocation visitor

```
public Object visitInvocation(Invocation inv) {
  Expression target = inv.target();
  Sequence params = inv.params();
  Name methodName = inv.methodName();
  TSType target_t = null;
  if (target != null) {
    target_t = (TSType) target.visit(this);
  } else {
    target_t = new Class(inv.myClass);
  }
  if (!target_t.isClass()) {
    System.out.println("Error: Attempt to invoked something
    ↪  not of class type");
  }
  Method invocation_t = null;
  java.util.List<TSType> params_t = new ArrayList<>();
  for (int i = 0; i < params.nchildren; ++i) {
    params_t.add((TSType) params.children[i].visit(this));
  }
  invocation_t = new Method(methodName.getname(), params_t,
  ↪  null);
  java.util.List<Method> candidates = findMethod(((Class)
  ↪  target_t).myDecl.allMethods, invocation_t, true);
  if (candidates.size() > 1) {
    System.out.println("More than one candidate remains: ");
    System.exit(1);
  } else if (candidates.size() == 0) {
    System.out.println("No candidates found!");
  } else {
    Method method = candidates.get(0);
    inv.setTSType(method.returnType);
    inv.targetMethod = (MethodDecl) method.myDecl;
  }
  System.out.println("> " + inv.getTSType());
  return (inv.getTSType() == null? inv.visit(this) :
  ↪  inv.getTSType());
}
```

For a method invocation $f(e_1, ..., e_n)$, we define a set of candidate methods (*matching*

names). This set is used to hold all candidate methods that have the correct name (f), the correct number of parameters, and the formal parameters that can hold the actual parameters with respect to assignment compatibility. We then start eliminating more general methods (from *possible candidates*), that is, a candidate g, whose parameters could be held in another candidate method's parameter. This results in the following specification and implementation

```
def findMethod(methods: Sequence, inv: Method,
 ↪  lookingForMethod:boolean) : List[Method] = {
  def matchingName:List[Method] <- List()
  forall m in 0..|{: methods.nchildren :}| : {
    def md:ClassBodyDecl <- {: methods.children[m] :} as
     ↪  ClassBodyDecl
    {: md.getname() :}.equals(inv.name) :} => matchingName +=
     ↪  {: md.visit(this) :}
  }
  def possibleCandidates:List[Method] <- List()
  forall m in matchingName : m :=T inv /\ m.name == inv.name
   ↪  => possibleCandidates += m
  forall i in 0..|possibleCandidates| : {
    def f:Method <- possibleCandidates[i]
    forall j in 0..|possibleCandidates| :
     ↪  possibleCandidates[j] :=T f => possibleCandidates -=
     ↪  possibleCandidates[j]
    possibleCandidates += f
  }
  return possibleCandidates
}
```

```
public  java.util.List<Method> findMethod(Sequence methods,
 ↪  Method inv, boolean lookingForMethod) {
  java.util.List<Method> matchingName = new ArrayList<>();
  for (int m = 0; m < methods.nchildren; ++m) {
    ClassBodyDecl md = (ClassBodyDecl) methods.children[m];
    if (md.getname().equals(inv.name)) {
      matchingName.add((Method) md.visit(this));
    }
```

```
  }
  java.util.List<Method> possibleCandidates = new
   ↪  ArrayList<>();
  for (var m : matchingName) {
    if (m.assignmentCompatible(inv) &&
     ↪  m.name.equals(inv.name)) {
      possibleCandidates.add(m);
    }
  }
  for (int i = 0; i < possibleCandidates.size(); ++i) {
    Method f = possibleCandidates.remove(0);
    for (int j = 0; j < possibleCandidates.size(); ++j) {
      if (possibleCandidates.get(j).assignmentCompatible(f)) {
        possibleCandidates.remove(possibleCandidates.get(j));
      }
    }
    possibleCandidates.add(f);
  }
  return possibleCandidates;
}
```

The combination of *visitInvocation* and *findMethod* is equivalent to the algorithm given in [5]. Algorithm 2 specifies the algorithm needed to determine the correct method to call.

Input: An invocation $I = f(e_1, ..., e_n)$
Output: A method to call or an error

1 **Function** *findMethod*
2 Let $T = (t_1, ..., t_n), t_i = \mathcal{T}(e_i)$
3 Let A be the set of all method with the same name as the one in the invocation and with the same number of parameters
4 Let $P = \{\}$
5 **foreach** $(f \in A)$ **do**
6 // $\mathcal{T}(f) = (p_1, ..., p_n) \rightarrow t$
7 **if** $\bigwedge_{i=1}^{n} p_i :=_{\mathcal{T}} t_i$ **then**
8 Add f to P
9 // $P = \{P_1, ..., P_m\}$ contains all candidates methods
10 **foreach** $(P_i, P_j \in P(i \neq j))$ **do**
11 **if** $P_i <_{\mathcal{T}} P_j$ **then**
12 // P_j is less specific than P_i so remove it from P
13 $P = P \backslash \{P_j\}$
14 **if** $|P| == 0$ **then**
15 // $P = \{\}$
16 Error("No method to call")
17 **else if** $|P| == 1$ **then**
18 // $P = \{P_k\}$, for some k $(1 \leq k \leq m)$
19 **return** P_k
20 **else**
21 // $|P| > 1$
22 $\exists a, b(a \neq b \wedge 1 \leq a, b \leq m) : P_a \not<_{\mathcal{T}} P_b \wedge P_b \not<_{\mathcal{T}} P_a$
23 Error("More than one possible method to call")

Algorithm 2: Find the correct method call algorithm.

The following bullet points explain the algorithm in detail as described in [5]:

- Let $I = f(e_1, ..., e_n)$ be the invocation.
- (Line 2) Let $T = (t_1, ..., t_n)$ be the type list of the invocation's actual parameters.
- (Line 3) Let A be the set of all methods with the same name as the one in the invocation, and with the same number of parameters.
- (Line 4) Let $P = \{\}$.

- (Lines 5-8) For each $f \in A$: where $\mathcal{T}(f) = (p_1, ..., p_n) \rightarrow t$ if $\bigwedge_{i=1}^{n} p_i :=_{\mathcal{T}} t_i$ then add f to P (f's formal parameters can hold the actual parameters).

- (Line 9) $P = \{P_1, ..., P_m\}$ now contains all the possible methods that could be a candidate to be the one called.

- (Lines 10-13) Consider each pair (P_i, P_j) where $i \neq j$: if $P_i <_{\mathcal{T}} P_j$ and P_j is still in P, then remove P_j from P. That is, if we considered P_i's parameter list to be a list of actual parameters, then they would be assignable to the formal parameters of P_j. Since P_i and P_j both are candidate methods for the invocation I, we can conclude that P_j is a **more general** (or less specific) method than P_i and therefore can be discarded as further candidate.

- (Lines 14-16) There are no methods that can be called for I; therefore, an error can be returned.

- (Lines 17-19) If $|P| == 1$ and $P = \{P_k\}$ for some value for k ($1 \leq k \leq m$), P_k is the method that will be called.

- (Lines 20-23) There are at least two candidate methods, P_a and P_b, that cannot eliminate each other because neither $P_a <_{\mathcal{T}} P_b$ nor $P_b <_{\mathcal{T}} P_a$, and no method can be determined as the one to call.

6.6 Simplicity vs. Expressiveness

To put the current comparisons into context, consider comparing a typed λ-calculus rule for type checking a binary expression with one written in *Jiapi*. Recall that, in typed λ-calculus, to type check an expression $E_1 \odot E_2$, we must try to construct a derivation of the judgment $\Gamma \vdash E_1 \odot E_2 : \mathcal{T}$, where T represents the type after computing the result of the binary operator $\odot$. Note that the $\odot$ operator divides the binary expression into two groups: the numeric ones and the relational ones (see Figure 21 on page 104). A rule in typed λ-calculus for type checking binary expressions in Espresso could look like the one given in Figure 44.

$$\frac{\mathbf{\Lambda}, C \vdash E_1 : \texttt{int} \quad \mathbf{\Lambda}, C \vdash E_2 : \texttt{int} \quad \ominus \in \{=, \neq, <, \leq, >, \geq\}}{\mathbf{\Lambda}, C \vdash E_1 \odot E_2 : \texttt{boolean}}$$

Figure 44. λ-calculus rule for type checking a binary expression.

Given the binary expression $a < 5$, we need to demonstrate that it can be expressed as a tree, where the root of the tree is the entire expression, each node is an instance of a typing rule, and leaves are rules that do not have a hypothesis. As we have seen earlier in the thesis, this is a relatively simple process. Keep in mind that we create this tree under the assumption that the variables have the types specified by C, so the expressions E_1 and E_2 are likely to have the types T_1 and T_2, respectively. For literals, however, we can use the following rule to infer their types: $\mathbf{\Lambda}, C \vdash n : \texttt{int}$.

In *Jiapi*, due to the expressiveness of the menta-language, we typically write difference cases when we specify type checking actions for binary operators because of the difference in behavior of the parse tree nodes. The easiest way to achieve this when coding is to switch on the oeprator. The description is given below

BinaryExpr

Syntactic Node:
BinaryExpr: be

Accessors:
$BinaryExpr\ left = be.left()$
$BinaryExpr\ right = be.right()$
$BinOp\ op = be.op()$

Rules:
$(Error_?(\mathcal{T}(left)) \vee Error_?(\mathcal{T}(right))) \Rightarrow$
$\quad \mathcal{T}(be) \leftarrow_{\mathcal{T}} Error$
| $(\{:\ \texttt{op.kind} == \texttt{BinOp.LT}\ :\} \vee \{:\ \texttt{op.kind} == \texttt{BinOp.LTEQ}\ :\} \vee$
$\quad \{:\ \texttt{op.kind} == \texttt{BinOp.GT}\ :\} \vee \{:\ \texttt{op.kind} == \texttt{BinOp.GTEQ}\ :\}) \Rightarrow$
$\quad (Numeric_?(\mathcal{T}(left)) \wedge Numeric_?(\mathcal{T}(right)) \Rightarrow$
$\quad\quad \mathcal{T}(be) = \lceil \mathcal{T}(left), \mathcal{T}(right) \rceil_{\mathcal{T}}$

| $\mathcal{T}(be) \leftarrow_{\mathcal{T}} Error(\text{"..."})$

Checks:

- Both $expr_1$ and $expr_2$must be of numeric type
- $\mathcal{T}(expr_1 \odot expr_2) \leftarrow_{\mathcal{T}} Boolean$

following this is its implementation

```
action BinaryExpr for be {
  left: Expression <- {: be.left() :}
  right: Expression <- {: be.right() :}
  op: BinOp <- {: be.op() :}
  Error?(T(left) \/ Error?(T(right))) =>
     T(be) <- Error
  , {: op.kind == BinOp.LT :} \/ {: op.kind == BinOp.LTEQ :}
   ↪  \/
    {: op.kind == BinOp.GT :} \/ {: op.kind == BinOp.GTEQ :}
     ↪  =>
      (Numeric?(T(left)) /\ Numeric?(T(right))) =>
         T(be) <- ceil(T(left), T(right))
       , T(be) <- Error
  ...
  return ...
}
```

How do we then verify that what we have is correct? In other words, can we create a tree in such a way that it demonstrates that our specification is correct? The answer is, of course, *yes*. We can demonstrate this using Tableau.

Example 10 (Constructing a proof): Suppose that $BinOp(o) : o \in \{<, >, \leq, \geq\}$ is a predicated used to determine whether the operator *op* is one of these: $<$, $>$, $\leq$, and $\geq$. Also, suppose that $Numeric(n)$ is another predicate that checks whether n is an numeric type, and $Apply(x, y)$ represents the operator $\lceil \alpha, \beta \rceil_{\mathcal{T}}$ (i.e., the ceiling function). We can use these predicates to write the following formula

$$\forall x \forall y \exists o(Binop(o) \wedge (Numeric(x) \wedge Numeric(y)) \wedge Apply(x, y)) \Rightarrow$$

$$\exists x \exists y(Apply(x, y))$$

which says "for all x and y there is an operator o such that if o is $<$, $>$, $\leq$, or $\geq$ and it can be applied to x and y and x and y are numbers, then there exists some x and y that o can be applied to." Now, to demonstrate the correctness of our rule for type checking binary expressions, we must prove the following formula. We assume it is false and look for a contradiction. So, let us begin with the signed formula contradiction. Note that due to space constrains, we will abbreviate the predicates as $N = \mathit{Numeric}$, $B = \mathit{BinOp}$, and $A = \mathit{Apply}$.

1.	$\neg(\forall x \forall y \exists a(B(a) \wedge (N(x) \wedge N(y)) \wedge A(x,y)) \Rightarrow \exists x \exists y(A(x,y)))$	
2.	$\forall x \forall y \exists a(B(a) \wedge (A(x,y) \wedge (N(x) \wedge N(y)))$	(1)
3.	$\neg \exists x \exists y(A(x,y))$	(1)
4.	$\forall y \exists a(B(a) \wedge (A(b,y) \wedge (N(x) \wedge N(y))))$	(2)
5.	$\exists a(B(a) \wedge (A(b,b) \wedge (N(b) \wedge N(b))))$	(4)
6.	$B(c) \wedge (A(b,b) \wedge (N(b) \wedge N(b)))$	(5)
7.	$B(c)$	(6)
8.	$A(b,b) \wedge (N(b) \wedge N(b))$	(6)
9.	$A(b,b)$	(8)
10.	$N(b) \wedge N(b)$	(8)
11.	$\neg \exists y A(b,y)$	(3)
12.	$\neg A(b,b)$	(11)
	closed!	

As we can see, despite some similarities (except for the quantifiers) between the two versions of the same rule, the generated tree (i.e., the derivation) differs due to the nature of tableau. Tableau not only shows that the formula is not a tautology, but it also provides a counterexample, that is, an interpretation of the variables that makes the formula false [239].

□

While we could still verify that our rule was correct, we had to first transform the components of the rule into predicates and then use these predicates to arrive to a conclusion. But could we ever get a derivation that is either similar to or the same as the one from typed λ-calculus? Unfortunately, the answer is **no**, and we will demonstrate this next.

Example 11: Consider the subtyping rule for a language such as Espresso, written in typed λ-calculus (left) and in *Jiapi*(right).

$$\frac{A \prec_{\mathcal{T}} B \quad B \prec_{\mathcal{T}} C}{A \prec_{\mathcal{T}} C} \qquad \forall x \forall y \forall z (x \prec_{\mathcal{T}} y \wedge y \prec_{\mathcal{T}} z) \Rightarrow \exists x \exists z (x \prec_{\mathcal{T}} z)$$

Can we prove the above implication? that is, can we demonstrate that if A is a subtype of B and B is a subtype of C, then A is also a subtype of C. In typed λ-calculus this would be as simple as $A \prec_{\mathcal{T}} B \wedge B \prec_{\mathcal{T}} C \Rightarrow A \prec_{\mathcal{T}} C$; for *Jiapi*, however, we would need to rely on the quantifiers. Why? Because quantifiers are the "words" used to refer to *quantities*. Unfortunately, this means that when we use quantifiers, we will want to selectively instantiate quantifier formulas on the left and right, with the goal of closing (or killing!) off branches as quickly as possible. Clearly, and as evidenced by Example 10, this can be challenging.

The proof is as follows

1.	$\forall x \forall y \forall z (x \prec_{\mathcal{T}} y \wedge y \prec_{\mathcal{T}} z)$	
2.	$\neg(\exists x \exists z (x \prec_{\mathcal{T}} z))$	
3.	$\exists y \exists z (a \prec_{\mathcal{T}} y \wedge y \prec_{\mathcal{T}} z)$	(1)
4.	$\exists z (a \prec_{\mathcal{T}} b \wedge b \prec_{\mathcal{T}} z)$	(3)
5.	$a \prec_{\mathcal{T}} b \wedge b \prec_{\mathcal{T}} c$	(4)
6.	$a \prec_{\mathcal{T}} b$	(5)
7.	$b \prec_{\mathcal{T}} c$	(5)
8.	$\neg \exists z (b \prec_{\mathcal{T}} z)$	(2)
9.	$\neg(b \prec_{\mathcal{T}} c)$	(8)
	closed!	

□

6.7 Conclusion

The specifications we developed in this section demonstrate that it is feasible to use set notation along with first-order logic to develop a type system for a DSL. However, we would like to highlight two observations that support *Jiapi*'s independent development:

- *Jiapi*'s parser generator only accepts grammars that extend meta-logic, which is useful for mathematical notations. The abstract syntax made it easier to encode bound names, but by fixing the grammar rules, it created a serious issue with type inference for domain-specific languages.

- Despite having simple syntax-based rules that can be extended to first-order logic, tableau has some theoretical flaws [240, 239]. For example, a rule may be deliberately applied, allowing for very efficient but also very inefficient non-terminating proofs. Another factor to consider is the tableau's branching. Because each branch must be worked on individually to check for closure, a large number of branches leads to a larger search space. Aside from the branching, the rules themselves allow for multiple appearances of the same formula.

Chapter 7

Related Work

7.1 Tools

We look at some of the existing relevant work in this chapter, then we explain the differences between them as well as the benefits and drawbacks of each.

7.1.1 CENTAUR and TYPOL

The CENTAUR [241] system is a generic interactive environment that, when provided with the description of a particular programming language, produces a language-specific environment; in other words, syntactic editors and semantic tools, such as type checkers or interpreters for a given language, can be automatically generated from the syntax and semantics specifications. It features a parser generator (*METAL* [242]) that converts a program's textual form into an AST. A pretty-printer written in *PPML* [243] (an abstract data type for dealing with a large number of sorted ASTs) converts the program's structural representation − which is already saved as an AST − into a concrete layout, and the logical engine is handled by TYPOL [244]. Here are a few instances of pretty-printed inference rules and axioms [245, Section 1.4]:

$$\frac{\texttt{env[] |- DECLS -> e \& e |- STMT}}{\texttt{|- \#program(DECLS, STMS)}} \qquad \frac{\rho_{\emptyset} \vdash \text{DECLS} : \rho \qquad \rho \vdash \text{STMS}}{\vdash \textbf{begin}\ \text{DECLS}\ \ \text{STMS}\ \textbf{end}}$$

$$\texttt{e |- decls[] -> e;} \quad \Leftrightarrow \quad \rho \vdash \text{decls}\,\texttt{[]} : \rho$$

$$\frac{\texttt{e |- DECL -> e1 \& e1 |- DECLS -> e2}}{\texttt{e |- decls[DECL.DECLS] -> e2;}} \qquad \frac{\rho \vdash \text{DECL} : \rho_1 \qquad \rho_1 \vdash \text{DECLS} : \rho_2}{\rho \vdash \text{DECL}\,;\text{DECLS} : \rho_2}$$

Figure 45. Inference rules and axioms (left) and pretty-printed (right).

TYPOL is a logical framework-based implementation of natural semantics [246] that has been used to describe type checking for a variety of PASCAL-like languages [244, Section 6]. A TYPOL specification is made up of an unordered set of inference rules. An inference rule with a numerator and denominator is the most fundamental unit of specification. Each inference rule has a numerator with a finite number of premises (which is empty for an axiom) and a denomiator with a conclusion. The premises and conclusion are represented by a Gentzen natural deduction sequence [247]. Sequents and conditions are the two types of formulae found in the premises. Although A can have a more generic structure, C must be a predicate. A sequent has the form $A \vdash C$ (a judgment) in which A is its *antecedent* and C is its *consequent*. Conditions are boolean expressions that can be defined either within TYPOL or through external procedures. Figure 46 illustrates an example of a TYPOL rule for expressing the dynamic semantics of the Eiffel [175] language's loop instruction:

```
SYSTEM, OBJL, CURROBJ, CURRMETH, BIND_PARAMS, RESULT
    |- <insts1> : OBJL1, RESULT1 &
expr_evaluation(SYSTEM,OBJL1,CURROBJ,BIND_PARAMS,RESULT1)
    |- <expr> : OBJL2, true())
-------------------------------------------------------------
SYSTEM, OBJL, CURROBJ, CURRMETH, BIND_PARAMS, RESULT
    |- from <insts1> until <expr> loop <insts2>: OBJL2, RESULT1 ;
```

Figure 46. Example of a TYPOL rule for expression dynamic semantics of Eiffel.

According to the foregoing criteria (as described in [248]), the instructions in the from section (<*insts*1>) must be executed first, followed by the evaluation of the expression <*expr*>. The loop is finished if it is true. (Note that another rule is needed in order to handle the *false* case.) It is worth noting that a TYPOL program's evaluation is an attempt to show

a goal $H_0 \vdash T_0 : S_0$ inside the logic given by the TYPOL program. A proof tree is created as a result of such an evaluation. The proof yields the semantic value, S_0, of the abstract syntax term, T_0, given some initial environment, H_0.

For execution, TYPOL specifications are compiled to Prolog. Prior to translation, the existing syntax trees are type checked using the formalism of operators, and variable types are deduced. Moreover, predicates' arguments are divided into two categories: *in* and *out* (known as argument kinds). To eliminate ambiguity in the resultant Prolog program, the types and kinds of a predicate's arguments are employed for overload resolution. The best matching choice is placed first, and the more precise rules outweigh the more general rules.

7.1.2 LATOS

LATOS [249] is a lightweight operational semantics animation tool that takes a Miranda [250] superset as input and generates a declarative program that can be type checked and run by the relevant language system. LATOS can be compiled into Miranda, which can then be converted to Haskell. It can be used for non-deterministic specifications, with one caveat: An animation can only produce one of many possible outcomes. All other features of LATOS, such as type checking, source dependency checking and type setting are available for deterministic as well as non-deterministic specifications. Although other large and powerful systems, such as CENTAUR [241] and ASF+SDF [251], include rendering, type checking, execution, animation, etc., LATOS focuses on making the most of existing components and carefully selecting necessary functionality.

Miranda is the basis of the LATOS input language, and it was enhanced with the following constructs [249, Section 3]: A fairly general notation for expressing abstract syntax as an algebraic data-type, a notation for expressing relations in terms of axioms and rules of inference, the addition of basic set theoretic expressions, and a number of minor features such as the provision of various types of brackets and ways of generating LaTeX symbols. The tool relies heavily on the target to provide semantics for language components that both the target and LATOS share. This is a fundamental distinction between LATOS and other tools that fix the semantics of the entire specification language. Figure 47 demonstrates this in great detail.

Comparison rule:

$$\frac{\vdash \langle s_1, s\rangle \xrightarrow{1} s'; \vdash \langle s_2, s'\rangle \xrightarrow{1} s''}{\vdash \langle s_1; s_2, s\rangle \xrightarrow{1} s'';} \ [\mathrm{comp_{ns}}]$$

LATOS specification:

```
rule comp_ns =
|- <s_1, s> =1=> s', |- <s_2, s'> =1=> s''
-------------------------------------------
      |- <s_1 $Comp s_2, s> =1=> s'' ;
```

Miranda output:

```
rule_1 [(s_1 $Comp s_2, s)]
  = [s''], if non_empty t_1 /\ non_empty t_2
    where
    s' = last t_1 ;
    t_1 = rule_1 [(s_1,s)] ;
    s'' = last t_2 ;
    t_2 = rule_1 [(s_2,s)] ;
    ;
```

Figure 47. Transformation: rule $\xrightarrow{to}$ LATOS $\xrightarrow{to}$ Miranda.

7.1.3 ASF+SDF

ASF+SDF is a specification formalism that combines the algebraic specification formalism ASF with the syntax definition formalism SDF [252, 253, 254]. ([251] provides an overview). ASF+SDF is built on conditional equations for defining semantics and broad context-free grammars for describing syntax. In this approach, the syntax of a (new or existing) language can be easily described, as well as operations on programs written in that language, such as static type checking, interpretation, compilation, or transformation. A basic ASF module is made up of a many-sorted algebraic signature that declares sorts, constants, and functions, as well as a collection of variables declarations and a set of positive conditional equations (see Figure 48). An ASF specification is a collection of named, perhaps parameterized modules with clear imports. On the other hand, a SDF specification is made up of productions that are comparable to those in EBNF but are reversed: $x_1..x_n \rightarrow s$, with each x_i representing

either a sort symbol (corresponding to a nonterminal symbol of the grammar) or a keyword or separator, and s representing a sort symbol. For example, a function with ASF signature, such as `f: s1#...#sn->s`, can be declared as `"f" "(" s1 ",".."," sn ")" -> s` in SDF.

```
module <ModuleName>
  <ImportSelection>*
  <ExportOrHiddenSection>*
    <Grammar>*
  equations
  <ConditionalEquation>*
```

Figure 48. The structure of an ASF module.

A prototype for the static type analysis of XQuery programs (a query language for the world-wide web) was introduced in [255]. The implementation was built using ASF+SDF, in which different semantics for commands were tested. Figure 49 presents part of the original definition of syntax and semantics of XQuery Core as it appears in [255]. The adaptation in ASF+SDF is also given in Figure 50.

$$\frac{\begin{array}{c}\textbf{statEnvs} \vdash Expr_1 : Type; \\ \textbf{statEnvs}[\textbf{varType}(Variable_1 : \textbf{prime}(Type_1))] \\ \vdash Expr_2 : Type_2\end{array}}{\begin{array}{c}\textbf{statEnvs} \vdash \textbf{for}\ Variable_1\ \textbf{in}\ Expr_1\ \textbf{return}\ Expre_2 : \\ Type_2.\textbf{quantifier}(Type_1)\end{array}}$$

$$\frac{\begin{array}{c}\textbf{statEnvs} \vdash Expr_1 : Type; \\ Type_0 = [SequenceType]_{sequencetype}; \\ Type_1 <: Type_0; \\ \textbf{statEnvs}[\textbf{varType}(Variable_1 : \textbf{prime}(Type_1))] \\ \vdash Expr_2 : Type_2\end{array}}{\begin{array}{c}\textbf{statEnvs} \vdash \textbf{for}\ SequenceType\ Variable_1\ \textbf{in}\ Expr_1 \\ \textbf{return}\ Expr_2 : Type_2.\textbf{quantifier}(Type_1)\end{array}}$$

Figure 49. The formal specification of *for* expressions.

When represented in the syntax of the ASF+SDF meta-environment, these rules are as follows:

```
[for-1] typecheck(E1, statEnvs) = T1,
     insert ($ Var1 --> prime(T1)) in
          VT of statEnvs = statEnvs1,
     typecheck(TExpr, statEnvs1) = T2
     ==========================================
     typecheck(for $ Var1 in E1 return TExpr,
          statEnvs = T2.quantifier(T1)

[for-2] typecheck(E1, statEnvs) = T1,
     T0 = normSeqT(SeqT),
     subtype(T1, T0, statEnvs) = true,
     insert($ Var1 --> prime(T1)) in
          VT of statEnvs = statEnvs1,
     typecheck(TExpr, statEnvs1) = T2
     ==========================================
     typecheck(for SeqT $ Var1 in E1 return TExpr,
          statEnvs = T2.quantifier(T1)

[default-for] typecheck(For, statEnvs) = static error
```

Figure 50. Adaptation in ASF+SDF.

Note that when no rule is applicable, the `[default-for]` rule is used, returning a static type error. Assuming now that there is a program containing a *for* expression to be checked, the function `typecheck` would be used in order to verify it. This function takes a program phrase and a (static) environment comprising all known variables and function and their types (see [255, Section 4]).

7.1.4 Tinker Type

TinkerType [58] is a language for controlling separate components of formal systems that are not necessarily type checkers, and then assembling them into a complete system using a set of options. Its formalist foundation is built on clauses (individual inference rules) and features that control their inclusion. For example, the rule below [58, Section 2]:

$$\frac{\Gamma;\Sigma \vdash t_1 : T_2 \to T_1 \qquad \Gamma;\Sigma \vdash t_2 : U \qquad U{<:}T_2}{\Gamma;\Sigma \vdash t_1 t_2 : T_1}\ \text{T-App}[\textit{arrow,sub,store}]$$

can be formalized by a set **Names** of clause names and a set **Cnt** of clause contents (an uninterpreted string), where

$\mathbf{Fts}$	a set of features
$\mathbf{Cls} \subseteq \mathbf{Names} \times \mathcal{P}(\mathbf{Fts}) \times \mathbf{Cnt}$	a set of clauses

with a feature dependency relation

$$\mathbf{Dep} \subseteq \mathcal{P}(\mathbf{Fts}) \times \mathcal{P}(\mathbf{Fts}) \quad \text{a dependency relation}$$

The $\mathit{closure}(F)$ is the least superset F' of F that is closed under the dependency relation for a given set of feature F. If $\mathit{closure}(F) \supseteq \mathit{closure}(F')$, then a set F dominates a set F'. A clause $\langle n, F, c\rangle$ – where $n \in \mathbf{Names}$, $F \subseteq \mathbf{Fts}$, and $c \in \mathbf{Cnt}$ – is eligible for inclusion if $\mathit{closure}(F_0) \supseteq \mathit{closure}(F)$. This means that the requested features include those that are covered by the clause (i.e., under the dependency relation). If more than one clause with the same name is eligible for inclusion, the clause with the most features (in terms of $\supseteq$) is selected; otherwise, an error is signaled. This technique ensures that the most relevant clause is selected.

The assembly of a type checker for a given set of features of a type system is one of TinkerType's uses. An ML clause for type checking conditional expressions, for example, is given in Figure 51 [58, Section 4].

```
T-If
  {#TmIf(fi,s1,s2,s3) ->
    if tyeqv ctx (typeof ctx s1) TyBool then
      let tyS = typeof ctx s2 in
      if tyeqv ctx tyS (typeof ctx s3) then tyS
      else error fi "arms of conditional have different types"
    else error fi "guard of conditional not a boolean"#}
```

Figure 51. `T-If` clause.

where `T-If` is the clause's name and the contents enclosed in `{#` and `#}` is a verbatim content. This segment computes the types of the guard expression `s1` and its two arms `s2`, `s3`, and

checks that `s1` has type *bool* and `s2`, `s3` are of the same type. The type of `s2` is the end outcome. It should be pointed out that all checks are module convertible, which is required in higher-order type system. TinkerType represents a range of checks that are comparable in their traditional implementation and may be constructed by textually concatenating single clauses. The feature dependencies act as consistency checks, determining whether clauses can or must be used together.

Chapter 8

Conclusion

8.1 Summary

This thesis described a formalism that we believe can be useful in type checking statically typed languages. We presented a method for automatically generating a type checker from type system descriptions using set notation and first-order logic. We were looking for this solution from the perspective of a typical compiler writer: someone who understands type systems but is not necessarily skilled at applying advanced techniques based on existing theorems.

Based on our analysis in Chapter 3, it is interesting to consider the historical development of type systems from the standpoint of a user. The fundamental form of type information has not changed; only the method by which it is obtained and used has. It is possible to construct an abstract interpretation of programs by viewing terms as objects with input and output, and then abstracting from the actual values those objects can have by looking only at what type they belong to. Type information improves program readability by providing additional, abstract (and thus less detailed) information about a program's structure. Additionally, type information is essential in the implementation during code generation: the information is required to obtain an efficient implementation as well as the ability to separate compilation of program modules.

In Chapter 4, we studied the mechanism of Espresso's type system; in particular, how we defined our typing rules and the structure of different data types, and how they can

be used together to construct proofs about the types of expressions (and statements if the language permits). However, merely having a type system specification is insufficient – we also need a type system that is effective! The goal is to demonstrate that a well-typed program never enters a"stuck state". In order to accomplish this, we must demonstrate progress and preservation to verify that our language is indeed type safe. For instance, an integer expression should not become a function after being "stepped". This is a prime example of a programming language's metatheoretical property [256]. That is, we prove the existence of specific types for expressions using the type system[51], and on the following step, we must demonstrate the existence of specific properties for our type system (or proof system) using structural induction to demonstrate type safety. This should provide a better understanding of the efforts that are required to write and validate a type system. Hence, it is an extremely challenging open research problem to automate such proofs.

With the present implementation of *Jiapi* (Chapter 5), it is possible to infer a type for any type. Types can be processed in any order, and creating a type requires no more user effort than declaring one. Unfortunately, due to the nature of the declarations, the inferred types usually exhibit bidericational behavior. Thus, the types of expression can be deduced not only from their arguments, but also from their usage. In general, the development of a type system in Jipai will follow the following path. For each clause declaration, (new) classes are created and pieced together with other classes (but only when necessary). Type inference is used in the early stages to infer general types with little regard for optimization. As the type system matures (i.e., more types are added), standard components (such as individual elements of a constructed type) are identified, and new class hierarchies (or relations) emerge. These class hierarchies are important in dealing with the analysis of the domain and range on which clauses are defined, as well as ensuring that these clauses are used appropriately after looking at the syntactic structure of many of the target language's constructs (not just its types).

Jiapi is applied to imperative and object-oriented languages in Chapter 6. In most cases, a language construct's type requirements can be captured directly in a single *Jiapi* rule

[51] The typing rule will provide us a more precise version of the proposition to prove in addition to particular facts from our inductive hypothesis.

(that is there is no need for programing the type checker). Because the typing rules are organized around language constructs, they have a high potential for reuse, which we take advantage of with C Minor's **until** statement. Furthermore, we show how to use tablaeu to construct proofs from existing rules, though this requires the use of predicates. As a result, the interpretation process is divided into atomic steps that are functions from one predicate to the next.

Overall, we believe that the ideas and concepts within the design of Jiapi have the potential support for turning complex type system into smaller rules in order to ensure type safety.

Appendix A

Background

A.1 First-order Logic (FOL)

First-order logic can be thought of as an extension of propositional logic [107, 257]. The goal of propositional logic is to establish mathematical truths by proving that statements hold. It is for this reason that atomic formulas in propositional logic have no internal structure since they are propositional variables that are either true or false. However, in first-order logic, they are predicates that assert a relationship between specific elements. Quantification is another important new concept here, which allows us the ability to assert that a certain property holds **for all** or **some** elements.

FOL's main feature is that it allows for the quantification of formulas over a given domain, leading to the introduction of universal and existential quantifiers. A formula φ containing a variable x is quantified using $\forall x$ to express that the formula will evaluate to true if x is replaced by any element of the domain. In contrast, existential (or $\exists x$) expresses that the formula will evaluate to true if x is replaced by some element of the domain. Furthermore, FOL supports predicates that describe properties of object (denoted conventionally with P, Q, ...), and functions that map objects to one another in a domain to express relations or characteristics.

A.1.1 Syntax

The syntax of first-order logic is made up of a collection of constant symbols, function symbols, and predicate symbols, referred to as a formula [257]. Each function and predicate symbols has an arity $n > 0$. In general, we use letters $a, b, ...$ to denote constant symbols, $f, g, ...$ to denote function symbols, and $P, Q, R, ...$ to denote predicate symbols. Thus, given a formula φ, the set of terms for φ is defined by the following inductive process [257]:

- Each variable is a term
- Each constant symbol is a term
- If $t_1, ..., t_n$ are terms and f is a function with arity $n > 0$, then $f(t_1, ..., t_n)$ is a term.

The set of formulas is defined inductively as follows:

1. Given terms $t_1, ..., t_n$ and a predicate symbol P with arity $n > 0$ then $P(t_1, ..., t_n)$ is a formula
2. For each formula F, $\neg F$ is also a formula
3. For each pair of formulas F and G, $F \vee G$ and $F \wedge G$ are both formulas
4. If F is a formula and x is a variable, then $\forall x F$ and $\exists x F$ are both formulas
5. Nothing else is a formula.

(1) tells us that $P(a_i)$ is a formula for any $i \in N$. These are the so called *atomic* formulas. They give us a starting point, whereas the other clauses show us how to create new formulas from existing ones. For example, by (2), we conclude that $\neg P(a_1)$ is a formula since $P(a_1)$ is already a formula by (1). (3) is a rather self-explanatory rule. In (4), we get that $\exists a_1 \neg P(a_1)$ is another formula, and so on. Finally, (5) tells us that **only** strings formed in this manner qualify as formulas.

Using these rules, we can encode the statement "Everyone likes ice cream" as $\forall x.likes(x, iceCream)$, or "There is no one who does not like ice cream" as $\neg\exists x.\neg likes(x, iceCream)$. In both of these cases, we need to be able to speak directly about objects (such as people or

numbers) and write down logical statements that generalize (or quantify) over those objects. This is made possible by first-order logic. Finally, the syntax of FOL allows us to explicitly represent objects and relationships between objects, giving us far more representational power than the prepositional case.

A.2 Set Notation

In mathematics, a set is a collection of objects (called elements or members) whose contents can be clearly defined [258]. Being well-defined means that a set has no ambiguity as to what objects are in the set or not. For example, the collection of all red laptops, the collection of positive numbers, etc, are all well-defined. The elements of a set are either written in **roster notation** (i.e., listing all the members of the set), or **set-builder notation** (i.e., telling how the set is created). For example, we write the set of whole numbers less than five in roster notation as $\{0, 1, 2, 3, 4\}$, whereas in set-notation we write it as $\{x \mid x \text{ is a whole number and } x < 5\}$.

A.2.1 Set-builder Notation

Set-builder notation is a precise way of expressing a collection of objects [258]. That is because we are either encoding or decoding mathematics when we use it. Although there is no standard format for set-builder notation, it does allow for some flexibility. Set-builder notation has the following representation

$$\{expression : rules\}$$

where the enclosing curly brackets denote a set, the vertical bar says "such that", and everything following the colon are conditions that explained the membership. We will now go over the operations used to manipulate sets, taking advantage of the opportunity to practice curly brackets notation.

Definition 12: The **empty set** is a set that contains zero elements. The empty set is denoted by $\{\}$ or $\emptyset$ □

Definition 13: We write $a \in A$ to denote that a is an element of the set A. $\in$ has a partner symbol $\notin$ that is used to say that an element is not in a set. For example, if $A = \{\{1\}, \{2, 3\}\}$, then $\{1\} \in A$, $1 \notin A$, but $1 \in \{1\}$. □

Definition 14: We say that two sets are equal if they contain **exactly** the same elements. For example, {red, green, blue} $\neq$ {r, g, b}, $\{1, \{2\}, 3\} \neq \{1, 2, 3\}$, but $\{1, \text{dog}, \psi\} = \{\psi,$ $\text{dog}, 1\}$. Note, order does not matter and repeated elements are superfluous; therefore, $\{\psi, \text{dog}, 1\} = \{\psi, \psi, 1, \{dog\}\}$. □

Definition 15: The size (or **cardinality**) of a set is the number of elements it contains. For example, for the set $\{1, \text{dog}, \psi\}$, we show cardinality by writing $|\{1, \text{dog}, \psi\}| = 3$. □

A.2.2 Operations

Now that we have established what sets are and how to think about them, let us categorize the operations that can be performed on them (but only the ones that we use in this thesis).

Definition 16: The **union** of two sets A and B is the collection of all objects that are in either set, and it is denoted by $A \cup B$. For example, $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$. Here the symbol "or" represents a disjunction, so if we think of A and B using set-builder notation like this

$$A = \{x \mid x \in A\}, \quad B = \{x \mid x \in B\}$$

then we end up with the original formula. For example, if $A = \{1, 2, 3\}$ and $B = \{4, 5\}$, then $A \cup B = \{1, 2, 3, 4, 5\}$. □

Definition 17: The **intersection** of two sets A and B is the collection of all object that are in both sets, and it is denoted by $A \cap B$. For example, $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$ means that if $A = \{1, 2, 3\}$ and $B = \{2, 3\}$, then $A \cap B = \{2, 3\}$. □

Definition 18: The **difference** of two sets A and B is the collection of all objects in A that are not in B, and it is denoted by $A \backslash B$ or $A - B$. For example, $A \backslash B = \{x \mid x \in A \text{ and } x \notin B\}$. □

Definition 19: For two sets A and B, A is a **subset** of B (denoted by $A \subset B$) if $\forall x.x \in A \Rightarrow x \in B$. More formally, any member of set A also is a member of set B; however, there may be members of B that are not in A (although A and B can be the same). For example, if $A = \{1, 2, 3\}$, then A has eight different subsets:

$$\begin{array}{cccc} \emptyset & \{1\} & \{2\} & \{3\} \\ \{1,2\} & \{1,3\} & \{2,4\} & \{1,2,3\} \end{array}$$

Notice that $A \subseteq A$ and in face each set is a subset of itself. (The empty set $\emptyset$ is a subset of every set.) □

A.2.3 Relations

In this section, we provide a brief overview of (binary) relations as they relate to compiling [5, 258]. Let X be a finite set of elements. A binary relation R on X is defined as a set of ordered pairs such that if $a \in X$ and $b \in X$, we write

$$a \sim b \Leftrightarrow (a, b) \in R$$

and say that "a is related to b". Generally speaking, a relationship on a set is denoted by the $\sim$ symbol, but other symbols that are more relevant to the context can be used. Let us look at the following relation examples (as given in [5, Example A.2]).

Example 12 (Relation): Let $X = \{1, 2, 3, ..., 10\}$ and the less-than operator $<$. We can create a relation R with respect to $<$ such that for $a \in X$ and $b \in X$:

$$a < b \Leftrightarrow (a, b) \in R$$

What does the relation (i.e., R) look like then? Since we understand how the $<$ operator works, we require every pair of elements in X in which the first element is less than the second. This gives us:

$$R = \{(1, 2), (1, 3), ..., (1, 10), (2, 3), ..., (2, 10), ..., (9, 10)\}$$

□

As can be seen, binary relations may themselves have properties. For example, a relation R on a set X and with relational operator $\sim$ is **transitive** if for $x, y, z \in X$

$$\text{if } x \sim y \wedge y \sim z \Rightarrow x \sim z$$

That is, if x is related to y and y is related to z, x is related to z.

Example 13 (Transitive Relation): The relation from Example 12 is transitive. For example, if we choose three numbers: 2, 5, and 5, then because $2 < 5$ and $5 < 7$ (and thus $(2, 5) \in R$ and $(5, 7) \in R$), $2 < 7$ and $(2, 7) \in R$. □

Furthermore, a relation R over a set X and with relational operator $\sim$ is **reflexive** if for all $x \in X$:

$$x \sim x$$

That is, x relates to itself.

Example 14 (Reflexive Relation): Using the same example (Example12), we can see that the relation R is no reflexive because no number x is strictly less than itself. However, if we had use the operator $\leq$ instead, we could have made the relation reflexive by addling all pairs $(x, x) \in R$. □

We summarize these relations here using FOL notation

Reflexivity: $\forall x R(x, x)$
Transitivity: $\forall x \forall y \forall z((R(x, y) \wedge R(y, z)) \rightarrow R(x, z))$

A.3 Tableau

Truth tables are based on the definition of a tautology as a formula that is true under all possible interpretations. Using truth tables to evaluate the truth of a formula, on the other hand, quickly becomes impractical. For larger formulas, we should look for a better approach, something that is fairly schematic but no longer requires checking individual valuations. Instead, it should be preferable to reason about truth and falsehood as such, and to analyze the condition for the truth of a formula in light of what we know about its sub-formulas.

Tableau is a schematic method that replaces the statement "X is true" with the signed formula TX and "X is false" with the signed formula FX [240, 239]. It replaces logical reasoning with the schematic application of syntactic manipulations to a signed formula, using the rules derived from the preceding observations. Before beginning with an example, let us look at the following observations [240]:

Remark 7: For all propositions X, Y:

1a. $T(\neg X) \Rightarrow FX$.

1b. $F(\neg X) \Rightarrow TX$.

2a. $T(X \wedge Y) \Rightarrow TX$ and TY.

2b. $F(X \wedge Y) \Rightarrow FX$ or FY.

3a. $T(X \vee Y) \Rightarrow TX$ or TY.

3b. $F(X \vee Y) \Rightarrow FX$ and FY.

4a. $T(X \rightarrow Y) \Rightarrow FX$ or TY.

4b. $F(X \rightarrow Y) \Rightarrow TX$ and FY.

□

The analytic tableau method can be summarized as follows: We assume FX and derive a contradiction using the rules from Remark 7 to prove the validity of a proposition X. In doing so, we adhere to a specific format, as shown in the following example.

Example 15 (Tableau Proof): An analytic tableau proving the validity of $\exists x \neg \phi(x) \rightarrow \neg \forall x \varphi(x)$ is shown next.

1.	$F\exists x \neg \varphi(x) \rightarrow \neg \forall x \varphi(x)$	Assumption

Since the main operator is $\rightarrow$, we apply the $\rightarrow F$ rule

1.	$F\exists x \neg \varphi(x) \rightarrow \neg \forall x \varphi(x)$ ✓	Assumption
2.	$T\exists x \neg \varphi(x)$	$\rightarrow F$ 1
3.	$F\neg \forall x \varphi(x)$	$\rightarrow F$ 1

The next line to solve is 2. We use $\exists T$. This requires a new constant symbols, and because no constant symbols have yet been created, we can choose any variable − say a

1.	$F\exists x\neg\varphi(x) \to \neg\forall x\varphi(x)\,\checkmark$	Assumption
2.	$T\exists x\neg\varphi(x)\,\checkmark$	$\to F\,1$
3.	$F\neg\forall x\varphi(x)$	$\to F\,1$
4.	$T\neg\varphi(x)$	$\exists T\,2$

Now we apply $\neg F$ rule to line 3

1.	$F\exists x\neg\varphi(x) \to \neg\forall x\varphi(x)\,\checkmark$	Assumption
2.	$T\exists x\neg\varphi(x)\,\checkmark$	$\to F\,1$
3.	$F\neg\forall x\varphi(x)\,\checkmark$	$\to F\,1$
4.	$T\neg\varphi(x)$	$\exists T\,2$
5.	$T\forall x\varphi(x)$	$\neg F\,3$

Applying $\neg T$ to line 4, followed by $\forall T$ to line 5, gives us a closed tableau.

1.	$F\exists x\neg\varphi(x) \to \neg\forall x\varphi(x)\,\checkmark$	Assumption
2.	$T\exists x\neg\varphi(x)\,\checkmark$	$\to F\,1$
3.	$F\neg\forall x\varphi(x)\,\checkmark$	$\to F\,1$
4.	$T\neg\varphi(x)$	$\exists T\,2$
5.	$T\forall x\varphi(x)$	$\neg F\,3$
6.	$F\varphi(a)$	$\neg T\,4$
7.	$T\varphi(a)$	$\forall T\,5$
	closed!	

We say that tableau method is said to have proved a formula X if there is a closed analytic tableau with origin FX. In other words, if a tableau branch contains a contradiction (i.e., a formula labeled with two different truth values), it is closed; otherwise, it is open. A completed open branch indicates the presence of an interpretation that fulfills the assignments at the tableau's root. A tableau with all branches closed demonstrates unsatisfiability and can be read as proof. □

A.3.1 Rules for the construction of Tableau

Let us go over what we just did. We used a schematic method that decomposes signed formulas based on the four observations about signed formulas (see Remark 7). Note that there are only two types of signed formulas [240]:

(A) $T(\neg X)$, $F(\neg X)$, $T(X \wedge Y)$, $F(X \vee Y)$, and $F(X \rightarrow Y)$ are five signed formulas with direct consequences

(B) $F(X \wedge Y)$, $F(X \vee Y)$, and $T(X \rightarrow Y)$ are three signed formulas that branch

When we use a formula of type (A), we simply add all of its direct consequences to each branch beneath the formula in question. In contrast to that, when using a formula of type (B), we divide each branch beneath the formula into two new branches. The rules for tableau are as follows:

$$\frac{T(\neg X)}{FX} \qquad \frac{F(\neg X)}{TX}$$

$$\frac{T(X \wedge Y)}{\begin{array}{c} TX \\ TY \end{array}} \qquad \frac{F(X \wedge Y)}{FX \mid FY}$$

$$\frac{T(X \vee Y)}{TX \mid TY} \qquad \frac{F(X \vee Y)}{\begin{array}{c} FX \\ FY \end{array}}$$

$$\frac{T(X \rightarrow Y)}{FX \mid TY} \qquad \frac{F(X \rightarrow Y)}{\begin{array}{c} TX \\ FY \end{array}}$$

Tableau can be written in a variety of styles. Each node can either write out the entire set of formulas or just the formula being analyzed. Truth value labels can be written explicitly, or they can be omitted and the false label can be replaced with the formula's negation. For classical propositional logic, the tableau method is a decision procedure; that is, a completely mechanical method that guarantees whether or not a formula is satisfiable. We can determine whether a given finite sequence is valid or not by starting with the assumption that all premises are true and the conclusion is false.

Bibliography

1. Andreas Stefik and Stefan Hanenberg. "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2014, pp. 283–299.
2. Charles A Hoare. *Hints on Programming Language Design*. Tech. rep. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1973.
3. Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. "Compilers, Principles, Techniques". In: *Addison wesley* 7.8 (1986), p. 9.
4. Luca Cardelli. "Type Systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.
5. Jan Bækgaard Pedersen. *Practical Compiler Construction: with Java and the JVM*. United States: Lulu.com, 2018. ISBN: 9781312519114.
6. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of computer security* 4.2-3 (1996), pp. 167–187.
7. John C Mitchell. "Type Systems for Programming Languages". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 365–458.
8. Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. "Array Bounds Check Elimination in the Context of Deoptimization". In: *Science of Computer Programming* 74.5-6 (2009), pp. 279–295.
9. Thi Viet Nga Nguyen and François Irigoin. "Alias Verification for Fortran Code Optimization". In: *J. UCS* 9.3 (2003), p. 270.
10. Jan Bækgaard Pedersen. *The Espresso Compiler*. `http://www.egr.unlv.edu/~matt/teaching/CSC460/Espresso.pdf`.
11. Jan Bækgaard Pedersen. *Process-Oriented Programming and Design with ProcessJ and CSP*. United States: Lulu.com, 2022. ISBN: 9781291220940.
12. Gregory J Duck and Roland HC Yap. "EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 181–195.
13. David May. "CSP, occam and Transputers". In: *Communicating Sequential Processes. The First 25 Years*. Springer, 2005, pp. 75–84.

14. Dick Pountain and David May. *A Tutorial Introduction to OCCAM Programming*. McGraw-Hill, Inc., 1987.

15. David May. "occam". In: *ACM Sigplan Notices* 18.4 (1983), pp. 69–79.

16. Peter H Welch and Fred RM Barnes. *Communicating mobile processes: Introducing occam-π. 25 Years of CSP (Lecture Notes in Computer Science, vol. 3525), Abdallah AE, Jones CB, Sanders JW*. 2005.

17. Peter H Welch and Frederick RM Barnes. "Communicating Mobile Processes". In: *Communicating Sequential Processes. The First 25 Years*. Springer, 2005, pp. 175–210.

18. Peter H Welch and Fred RM Barnes. "Mobile Barriers for occam-π: Semantics, Implementation and Application." In: *CPA*. Vol. 5. 2005, pp. 289–316.

19. Peter H Welch. "Life of occam-π". In: *Communicating Process Architectures 2013* (2013), pp. 293–318.

20. Bertil Svensson et al. "occam-πas a High-Level Language for Coarse-Grained Reconfigurable Architectures". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 236–243.

21. Jan Bækgaard Pedersen and Marc L Smith. "ProcessJ: A Possible Future of Process-Oriented Design." In: *CPA*. 2013, pp. 133–156.

22. *occam-π and KRoC: Blending CSP and the π-calculus*. `https://www.cs.kent.ac.uk/projects/ofa/kroc/`.

23. Muhammad Taimoor Khan and Wolfgang Schreiner. "Towards the Formal Specification and Verification of Maple Programs". In: *International Conference on Intelligent Computer Mathematics*. Springer. 2012, pp. 231–247.

24. Robert Sebesta. *Concepts of Programming Languages*. Boston: Pearson, 2016. ISBN: 978-0133943023.

25. Michael Scott. *Programming Language Pragmatics*. San Francisco, CA: Morgan Kaufmann Pub, 2006. ISBN: 978-0124104099.

26. Microsoft. *The C# Type System — Microsoft Docs*. `https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/`.

27. Klaus Ostermann. "Nominal and Structural Subtyping in Component-Based Programming." In: *J. Object Technol.* 7.1 (2008), pp. 121–145.

28. Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 471–523.

29. Luca Cardelli and John C Mitchell. "Operations on Records". In: *Mathematical structures in computer science* 1.1 (1991), pp. 3–48.

30. Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: *Information and Computation* 93.1 (1991), pp. 1–15.

31. Luca Cardelli. "A Semantics of Multiple Inheritance". In: *International symposium on semantics of data types*. Springer. 1984, pp. 51–67.

32. William R Cook, Walter Hill, and Peter S Canning. “Inheritance is not Subtyping”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 125–135.

33. John C Mitchell. “Toward a Typed Foundation for Method Specialization and Inheritance”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 109–124.

34. Uday Reddy. “Objects as Closures: Abstract Semantics of Object-Oriented Languages”. In: *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. 1988, pp. 289–297.

35. Andrew J Kennedy and Benjamin C Pierce. “On Decidability of Nominal Subtyping with Variance”. In: (2006).

36. Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. “Complete and Decidable Type Inference for GADTs”. In: *ACM Sigplan Notices* 44.9 (2009), pp. 341–352.

37. Aleksandr Misonizhnik and Dmitry Mordvinov. “On Satisfiability of Nominal Subtyping with Variance”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

38. Ori Roth. “Study of the Subtyping Machine of Nominal Subtyping with Variance (full version)”. In: *arXiv preprint arXiv:2109.03950* (2021).

39. Yuri Leontiev, M Tamer Özsu, and Duane Szafron. “On Type Systems for Object-Oriented Database Programming Languages”. In: *ACM Computing Surveys (CSUR)* 34.4 (2002), pp. 409–449.

40. Michael Kölling. “The Problem of Teaching Object-Oriented Programming, Part 1: Languages”. In: *Journal of Object-oriented programming* 11.8 (1999), pp. 8–15.

41. Michael Kölling. “The Problem of Teaching Object-Oriented Programming, Part 2: Environments”. In: *Journal of Object-Oriented Programming* 11.9 (1999), pp. 6–12.

42. Will Dietz, Peng Li, John Regehr, and Vikram Adve. “Understanding Integer Overflow in C/C++”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.1 (2015), pp. 1–29.

43. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Pearson Education, 2014.

44. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 2000.

45. Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

46. Jana Dunfield. “Annotations for Intersection Typechecking”. In: *arXiv preprint arXiv:1307.8204* (2013).

47. Groovy Language. *Type Checking Extensions*. `https://docs.groovy-lang.org/next/html/documentation/type-checking-extensions.html`.

48. *The Apache Groovy Programming Language — Runtime and Compile-Time Metaprogramming*. URL: http://www.groovy-lang.org/metaprogramming.html#_compile_time_metaprogramming.

49. Prodromos Gerakios, Aggelos Biboudis, and Yannis Smaragdakis. "Reified Type Parameters Using Java Annotations". In: *Proceedings of the 12th international conference on Generative programming: concepts & experiences*. 2013, pp. 61–64.

50. Michael D Ernst. *Type annotations specification (JSR 308)*. 2008.

51. Raymie Stata and Martin Abadi. "A Type System for Java Bytecode Subroutines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.1 (1999), pp. 90–137.

52. Stephen N Freund and John C Mitchell. "A Type System for the Java Bytecode Language and Verifier". In: *Journal of Automated Reasoning* 30.3 (2003), pp. 271–321.

53. Stephen N Freund and John C Mitchell. "A Type System for Object Initialization in the Java Bytecode Language". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.6 (1999), pp. 1196–1250.

54. Gilles Barthe, David Pichardie, and Tamara Rezk. "A Certified Lightweight Non-Interference Java Bytecode Verifier". In: *European Symposium on Programming*. Springer. 2007, pp. 125–140.

55. Benjamin C Pierce. *Advanced Topics in Types and Programming Languages*. MIT press, 2004.

56. Jan Van Leeuwen. *Handbook of Theoretical Computer Science (vol. A) Algorithms and Complexity*. Mit Press, 1991.

57. Thomas Reps. *Generating Language-Based Environments*. Tech. rep. Cornell University, 1982.

58. Michael Y Levin and Benjamin C Pierce. "Tinkertype: A Language for Playing with Formal Systems". In: *Journal of Functional Programming* 13.2 (2003), pp. 295–316.

59. Erik Meijer and Peter Drayton. "Static Typing where Possible, Dynamic Typing when Needed: The End of the Cold War Between Programming Languages". In: Citeseer. 2004.

60. Jan Herman Geuvers. *Logics and Type Systems*. [Sl: sn], 1993.

61. *Judgment in nLab*. https://ncatlab.org/nlab/show/judgment.

62. Luca Cardelli. *Type Systems*. http://lucacardelli.name/papers/typesystems.pdf.

63. Jens Palsberg and C Barry Jay. "The Essence of the Visitor Pattern". In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE. 1998, pp. 9–15.

64. Bruno CdS Oliveira, Meng Wang, and Jeremy Gibbons. "The Visitor Pattern as a Reusable, Generic, Type-Safe Component". In: *Proceedings of the 23rd ACM*

SIGPLAN conference on Object-oriented programming systems languages and applications. 2008, pp. 439–456.

65. UNLV. *CS 460 Compiler Construction — Acalog ACMS™*. `https://catalog.unlv.edu/preview_course_nopop.php?catoid=32&coid=159240`.

66. Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Vol. 9. Bibliopolis Naples, 1984.

67. Thierry Coquand. "Metamathematical Investigations of a Calculus of Constructions". PhD thesis. INRIA, 1989.

68. Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

69. Kazuo Thow. *The Curry-Howard isomorphism: of proofs and programs*. `https://sites.math.washington.edu/~morrow/336_11/papers/kazuo.pdf`.

70. J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and Combinators, an Introduction*. Vol. 2. Cambridge University Press Cambridge, 2008.

71. Dennis M Ritchie. "The Development of the C Language". In: *ACM Sigplan Notices* 28.3 (1993), pp. 201–208.

72. Bjarne Stroustrup. *The C++ Programming Language: Reference Manual*. Tech. rep. Bell Lab., 1984.

73. Margaret A Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

74. Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report: ISO PASCAL Standard*. Springer Science & Business Media, 2012.

75. Göran Sundholm. "Systems of Deduction". In: *Handbook of philosophical logic*. Springer, 1983, pp. 133–188.

76. Henk Barendregt. "Introduction to Generalized Type Systems". In: *Journal of functional programming* 1.2 (1991), pp. 125–154.

77. Ross T Brady. "Natural Deduction Systems for Some Quantified Relevant Logics". In: *Logique et Analyse* 27.108 (1984), pp. 355–377.

78. LS van Benthem Jutting, James McKinna, and Robert Pollack. "Checking Algorithms for Pure Type Systems". In: *International Workshop on Types for Proofs and Programs*. Springer. 1993, pp. 19–61.

79. Craig Chambers and Gary T Leavens. "Typechecking and Modules for Multimethods". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.6 (1995), pp. 805–843.

80. Mitchell Wand. "A Simple Algorithm and Proof for Type Inference". In: *Fundamenta Informaticae* 10.2 (1987), pp. 115–121.

81. Martin Odersky, Martin Sulzmann, and Martin Wehr. "Type Inference with Constrained Types". In: *Theory and practice of object systems* 5.1 (1999), pp. 35–55.

82. Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. "Constrained Types for Object-Oriented Languages". In: *Proceedings of the 23rd*

ACM SIGPLAN conference on Object-oriented programming systems languages and applications. 2008, pp. 457–474.

83. Valery Trifonov and Scott Smith. "Subtyping Constrained Types". In: *International Static Analysis Symposium*. Springer. 1996, pp. 349–365.

84. 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*. 1982, pp. 207–212.

85. Philip Wadler and Stephen Blott. "How to Make Ad-hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 60–76.

86. Robin Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

87. Didier Rémy. "Type Checking Records and Variants in a Natural Extension of ML". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 77–88.

88. Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. "Type Classes in Haskell". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), pp. 109–138.

89. Kung Chen, Paul Hudak, and Martin Odersky. "Parametric Type Classes". In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. 1992, pp. 170–181.

90. Martin Odersky, Philip Wadler, and Martin Wehr. "A Second Look at Overloading". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. 1995, pp. 135–146.

91. John C Mitchell. "Coercion and Type Inference". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1984, pp. 175–185.

92. John C Mitchell. "Type Inference with Simple Subtypes". In: *Journal of functional programming* 1.3 (1991), pp. 245–285.

93. Kim B Bruce, Angela Schuett, and Robert van Gent. "PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language". In: *European Conference on Object-Oriented Programming*. Springer. 1995, pp. 27–51.

94. Alexander Aiken and Edward L Wimmers. "Type Inclusion Constraints and Type Inference". In: *Proceedings of the conference on Functional programming languages and computer architecture*. 1993, pp. 31–41.

95. Stefan Hanenberg. "An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 22–35.

96. Kenneth Knowles and Cormac Flanagan. "Hybrid Type Checking". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32.2 (2010), pp. 1–34.

97. Cormac Flanagan. "Hybrid Type Checking". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2006, pp. 245–256.

98. Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. Vol. 32. Citeseer, 2007.

99. Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 214–227.

100. Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. "Sage: Hybrid Checking for Flexible Specifications". In: *Scheme and Functional Programming Workshop*. Vol. 6. 2006, pp. 93–104.

101. Zachary Ryan Anderson. *Static Analysis of C for Hybrid Type Checking*. Tech. rep. Tech. Rep. EECS-2007-1, UC Berkeley, 2007.

102. Edsger W Dijkstra. "Programming as a Discipline of Mathematical Nature". In: *The American Mathematical Monthly* 81.6 (1974), pp. 608–612.

103. Edsger W Dijkstra. "A Constructive Approach to the Problem of Program Correctness". In: *BIT Numerical Mathematics* 8.3 (1968), pp. 174–186.

104. Edsger W Dijkstra et al. "On the Cruelty of Really Teaching Computing Science". In: *Communications of the ACM* 32.12 (1989), pp. 1398–1404.

105. Philip L Frana and Thomas J Misa. "An Interview with Edsger W. Dijkstra". In: *Communications of the ACM* 53.8 (2010), pp. 41–47.

106. Edsger W Dijkstra. "The Humble Programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

107. Fairouz D Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From Its Origins Until Today*. Vol. 29. Springer Science & Business Media, 2004.

108. John C Mitchell. *Foundations for Programming Languages*. Vol. 1. MIT press Cambridge, 1996.

109. Brian Chin, Shane Markstrum, and Todd Millstein. "Semantic Type Qualifiers". In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 85–95.

110. *Type Theory (Stanford Encyclopedia of Philosophy)*. `https://plato.stanford.edu/entries/type-theory/`.

111. Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking through Dependent Types". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 249–257.

112. Dominic Duggan and John Ophel. "Type-Checking Multi-Parameter Type Classes". In: *Journal of functional programming* 12.2 (2002), pp. 133–158.

113. Martin Odersky and Konstantin Läufer. "Putting Type Annotations to Work". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1996, pp. 54–67.

114. Benjamin C Pierce and C Benjamin. *Types and Programming Languages*. MIT press, 2002.

115. Gianluigi Bellin, Valeria De Paiva, and Eike Ritter. "Extended Curry-Howard Correspondence for a Basic Constructive Modal Logic". In: *Proceedings of methods for modalities*. Vol. 2. 2001.

116. John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. "Report on The algorithmic Language ALGOL 60". In: *Communications of the ACM* 3.5 (1960), pp. 299–314.

117. Ole-Johan Dahl and Kristen Nygaard. "SIMULA: An ALGOL-based Simulation Language". In: *Communications of the ACM* 9.9 (1966), pp. 671–678.

118. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. "An Overview of the Scala Programming Language". In: (2004).

119. Guido VanRossum and Fred L Drake. *The Python Language Reference*. Python Software Foundation Amsterdam, Netherlands, 2010.

120. *The Perl Programming Language - www.perl.org*. `https://www.perl.org/`.

121. *Functions - JavaScript — MDN*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions`.

122. David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. "TIL: A Type-Directed Optimizing Compiler for ML". In: *ACM Sigplan Notices* 31.5 (1996), pp. 181–192.

123. J. Oberg. "Why the Mars probe went off course [accident investigation]". In: *IEEE Spectrum* 36.12 (1999), pp. 34–39. DOI: 10.1109/6.809121.

124. Malaya Kumar Biswal M and Ramesh Naidu Annavarapu. "A Study on Mars Probe Failures". In: *AIAA Scitech 2021 Forum*. 2021, p. 1158.

125. Chung Y Lo. "NASA's Space-Probes Pioneer Anomaly and the Mass-Charge Repulsive Force". In: *17th Annual Natural Philosophy Alliance Conference, California State University, Long Beach, California June*. 2010, pp. 23–26.

126. Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 155–165.

127. Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. "A Large-Scale Empirical Study of Compiler Errors in Continuous Integration". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 176–187.

128. Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. "An Empirical Study on the Impact of Static Typing on Software Maintainability". In: *Empirical Software Engineering* 19.5 (2014), pp. 1335–1382.

129. Stefan Hanenberg. "Costs of Using Untyped Programming Languages–First Empirical Results". In: *IFAC Proceedings Volumes* 42.4 (2009), pp. 1418–1422.

130. Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M Kent, Vincent St-Amour, T Stephen Strickland, and Asumu Takikawa. "Migratory Typing: Ten Years Later". In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

131. Jukka Lehtosalo. *Our Journey to Type Checking 4 Million Lines of Python — Dropbox*. `https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python`.

132. Faizan Khan, Boqi Chen, Daniel Varro, and Shane Mcintosh. "An Empirical Study of Type-Related Defects in Python Projects". In: *IEEE Transactions on Software Engineering* (2021).

133. IDEA IntelliJ. "The Most Intelligent Java IDE". In: JetBrains [online].[cit. 2016-02-23]. Dostupné z: `https://www.jetbrains.com/idea/#chooseYourEdition` (2011).

134. Ed Burnette. *Eclipse IDE Pocket Guide: Using the Full-Featured IDE*. "O'Reilly Media, Inc.", 2005.

135. Apache NetBeans. *Welcome to Apache NetBeans*. `https://netbeans.apache.org/`.

136. Jens Palsberg. "Type-Based Analysis and Applications". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2001, pp. 20–27.

137. William R Bush, Jonathan D Pincus, and David J Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors". In: *Software: Practice and Experience* 30.7 (2000), pp. 775–802.

138. Cindy Rubio-González and Ben Liblit. "Finding Error-Handling Bugs in Systems Code Using Static Analysis". In: *PhD Forum of the Grace Hopper Celebration of Women in Computing, Portland, Oregon*. 2011.

139. Urs Hölzle and Ole Agesen. "Dynamic versus Static Optimization Techniques for Object-Oriented Languages". In: *Theory and Practice of Object Systems* 1.3 (1995), pp. 167–188.

140. Jan Bækgaard Pedersen and Brian Kauke. "Resumable Java Bytecode-Process Mobility for the JVM." In: *CPA*. 2009, pp. 159–172.

141. Oswaldo Benjamin Cisneros Merino. "ProcessJ: The JVMCSP Code Generator". In: (2019).

142. Alexander C Thomason. "The ProcessJ C++ Runtime System and Code Generator". PhD thesis. University of Nevada, Las Vegas, 2020.

143. Matthew Sowders. "ProcessJ: A Process-Oriented Programming Language". In: (2011).

144. Fraser Brown, Andres Nötzli, and Dawson Engler. "How to Build Static Checking Systems Using Orders of Magnitude Less Code". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 143–157.

145. Andrew K Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and computation* 115.1 (1994), pp. 38–94.

146. Alfred North Whitehead and Bertrand Russell. *Principia Mathematica to* 56*. Vol. 2. Cambridge University Press, 1997.

147. Frank Plumpton Ramsey. "The Foundations of Mathematics". In: (1925).

148. Alonzo Church. "A Formulation of the Simple Theory of Types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

149. Stefano Berardi. "Towards a Mathematical Analysis of the Coquand-Huet Calculus of Constructions and the Other Systems in Barendregt's Cube". In: *Technica1 report, Carnegie-Me11on University (USA) and Universita di Torino (Ita1y)* (1988).

150. Richard A De Millo, Richard J Lipton, and Alan J Perlis. "Social Processes and Proofs of Theorems and Programs". In: *Communications of the ACM* 22.5 (1979), pp. 271–280.

151. Jonathan Bowen and Victoria Stavridou. "Safety-Critical Systems, Formal Methods and Standards". In: *Software engineering journal* 8.4 (1993), pp. 189–209.

152. Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, Inc., 1990.

153. Peter D Mosses. "The Varieties of Programming Language Semantics and Their Uses". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2001, pp. 165–190.

154. Dana Scott. "Mathematical Concepts in Programming Language Semantics". In: *Proceedings of the May 16-18, 1972, spring joint computer conference*. 1971, pp. 225–234.

155. Tobias Nipkow. "Programming and Proving in Isabelle/HOL". In: *Technical report, University of Cambridge*. 2013.

156. Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

157. Harry R Lewis and Christos H Papadimitriou. "Elements of the Theory of Computation". In: *ACM SIGACT News* 29.3 (1998), pp. 62–78.

158. Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques and Tools*. 2020.

159. *Analytica Wiki*. `https://wiki.analytica.com/index.php?title=Analytica_Wiki&title=Analytica_Wiki`.

160. Joseph A Bank, Andrew C Myers, and Barbara Liskov. "Parameterized types for Java". In: *Proceedings of the 24th acm sigplan-sigact symposium on principles of programming languages*. 1997, pp. 132–145.

161. Brian McNamara and Yannis Smaragdakis. "Static Interfaces in C++". In: *First Workshop on C++ Template Programming*. Citeseer. 2000, pp. 26–31.

162. David Vandevoorde. *C++ Templates : The Complete Guide*. Boston: Addison-Wesley, 2018. ISBN: 978-0321714121.

163. *C++ International Standard*. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf`.

164. Martin Richards. "BCPL: A Tool for Compiler Writing and System Programming". In: *Proceedings of the May 14-16, 1969, spring joint computer conference*. 1969, pp. 557–566.

165. *N1570 April 12, 2011 ISO/IEC 9899:201x*. URL: `http://port70.net/~nsz/c/c11/n1570.html#I`.

166. Allen B Tucker. *Computer Science Handbook*. CRC press, 2004.

167. David Gries and Narain Gehani. "Some Ideas on Data Types in High-Level Languages". In: *Communications of the ACM* 20.6 (1977), pp. 414–420.

168. Thomas A Standish. "Extensibility in Programming Language Design". In: *Proceedings of the May 19-22, 1975, national computer conference and exposition*. 1975, pp. 287–290.

169. Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2008.

170. Kim B Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT press, 2002.

171. Kim B Bruce. "Typing in Object-Oriented Languages: Achieving Expressiveness and Safety". In: *Unpublished, June* (1996).

172. Benjamin Evans. *Java in a Nutshell*. Beijing, China Boston, Massachusetts: O'Reilly Media, Inc, 2018. ISBN: 9781492037255.

173. Bjarne Stroustrup. *The C++ Programming Language*. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 978-0321563842.

174. Microsoft. *The C# Type System – Microsoft Docs*. `https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/`.

175. Bertrand Meyer. *Eiffel: The Language*. New York: Prentice Hall, 1992. ISBN: 978-0132479257.

176. Venkat Subramaniam. *Programming Groovy 2 : Dynamic Productivity for the Java Developer*. Frisco, TX: The Pragmatic Programmers, 2014. ISBN: 9781937785307.

177. JetBrains. *Type System – Kotlin Language Specification*. `https://kotlinlang.org/spec/type-system.html`.

178. Gianluca Mezzetti, Anders Møller, and Fabio Strocco. "Type Unsoundness in Practice: An Empirical Study of Dart". In: *ACM SIGPLAN Notices* 52.2 (2016), pp. 13–24.

179. Per Runeson. "A Survey of Unit Testing Practices". In: *IEEE software* 23.4 (2006), pp. 22–29.

180. Michael Olan. "Unit Testing: Test Early, Test Often". In: *Journal of Computing Sciences in Colleges* 19.2 (2003), pp. 319–328.

181. Yoonsik Cheon and Gary T Leavens. "A Simple and Practical Approach to Unit Testing: The JML and JUnit way". In: *European Conference on Object-Oriented Programming*. Springer. 2002, pp. 231–255.

182. Qing Li and Yu-Liu Chen. "Data Flow Diagram". In: *Modeling and Analysis of Enterprise and Information Systems*. Springer, 2009, pp. 85–97.

183. Alexandru G Bardas et al. "Static Code Analysis". In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107.

184. Jongwook Woo, Isabelle Attali, Denis Caromel, J-L Gaudiot, and Andrew L Wendelborn. "Alias Analysis on Type Inference for Class Hierarchy in Java". In: *Proceedings 24th Australian Computer Science Conference. ACSC 2001.* IEEE. 2001, pp. 206–214.

185. Maryam Emami. "A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing C Compiler". PhD thesis. McGill University Montreal, Québec, 1993.

186. Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. "A Compiler Framework for Speculative Analysis and Optimizations". In: *ACM SIGPLAN Notices* 38.5 (2003), pp. 289–299.

187. Philippas Tsigas and Yi Zhang. "A Simple, Fast Parallel Implementation of Quicksort and Its Performance Evaluation on SUN Enterprise 10000". In: *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.* IEEE. 2003, pp. 372–381.

188. Philip Heidelberger, Alan Norton, and John T. Robinson. "Parallel Quicksort Using Fetch-and-Add". In: *IEEE Transactions on Computers* 39.1 (1990), pp. 133–138.

189. Daniel Cederman and Philippas Tsigas. "Gpu-quicksort: A Practical Quicksort Algorithm for Graphics Processors". In: *Journal of Experimental Algorithmics (JEA)* 14 (2010), pp. 1–4.

190. *Warning Options (Using the GNU Compiler Collection (GCC)).* `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`.

191. Giuseppe Penna. "A Type System for Static and Dynamic Checking of C++ Pointers". In: *Computer Languages, Systems & Structures* 31 (July 2005), pp. 71–101. DOI: `10.1016/j.cl.2004.05.002`.

192. Gray Watson. "Debug Malloc Library". In: *Letters Corp* 11 (1994).

193. Gene Novark, Emery D Berger, and Benjamin G Zorn. "Exterminator: Automatically Correcting Memory Errors with High Probability". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 1–11.

194. Glenn R Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li, Olga Taborskaia, and Yanmei Wang. "A Survey of Systems for Detecting Serial Run-time Errors". In: *Concurrency and Computation: Practice and Experience* 18.15 (2006), pp. 1885–1907.

195. Scott Milton, Heinz Schmidt, et al. "Dynamic Dispatch in Object-Oriented Languages". In: (1994).

196. Craig Chambers. "Object-Oriented Multi-Methods in Cecil". In: *European Conference on Object-Oriented Programming*. Springer. 1992, pp. 33–56.

197. Eric Amiel, Olivier Gruber, and Eric Simon. "Optimizing Multi-method Dispatch Using Compressed Dispatch Tables". In: *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*. 1994, pp. 244–258.

198. David Wetherall and Christopher J Lindblad. "Extending Tcl for Dynamic Object-Oriented Programming." In: *Tcl/Tk Workshop*. Vol. 670. 1995.

199. Mary F Fernandez. "Simple and Effective Link-Time Optimization of Modula-3 Programs". In: *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. 1995, pp. 103–115.

200. Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". In: *European Conference on Object-Oriented Programming*. Springer. 1995, pp. 77–101.

201. John Plevyak and Andrew A Chien. "Precise Concrete Type Inference for Object-Oriented Languages". In: *ACM SIGPLAN Notices* 29.10 (1994), pp. 324–340.

202. Gosling James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification – Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

203. Thierry Coquand. "An Algorithm for Type-Checking Dependent Types". In: *Science of Computer Programming* 26.1-3 (1996), pp. 167–177.

204. Satish Chandra and Thomas Reps. "Physical Type Checking for C". In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1999, pp. 66–75.

205. Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. "Type Checking with Open Type Functions". In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 2008, pp. 51–62.

206. Jens Palsberg and Michael I Schwartzbach. "Object-Oriented Type Inference". In: *ACM SIGPLAN Notices* 26.11 (1991), pp. 146–161.

207. Dominic Duggan and Frederick Bent. “Explaining Type Inference”. In: *Science of Computer Programming* 27.1 (1996), pp. 37–83.

208. BJ Heeren, Jurriaan Hage, S Doaitse Swierstra, et al. “Generalizing Hindley-Milner Type Inference Algorithms”. In: (2002).

209. David McAllester. “A Logical Algorithm for ML Type Inference”. In: *International Conference on Rewriting Techniques and Applications*. Springer. 2003, pp. 436–451.

210. 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*. 2006, pp. 167–178.

211. Roberto Barbuti and Roberto Giacobazzi. “A Bottom-Up Polymorphic Type Inference in Logic Programming”. In: *Science of computer programming* 19.3 (1992), pp. 281–313.

212. Apple. *About Swift — The Swift Programming Language (Swift 5.6)*. `https://docs.swift.org/swift-book/`.

213. *The Python Language Reference — Python 3.10.4 documentation*. `https://docs.python.org/3/reference/index.html`.

214. *mypy - Optional Static Typing for Python*. `http://mypy-lang.org/`.

215. *PyType - A Static Type analyzer for Python Code*. `https://google.github.io/pytype/`.

216. *PEP 484 – Type Hints — peps.python.org*. `https://peps.python.org/pep-0484/`.

217. Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. “Python 3 Types in the Wild: A Tale of Two Type Systems”. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. 2020, pp. 57–70.

218. *PyCharm: the Python IDE for Professional Developers by JetBrains*. `https://www.jetbrains.com/pycharm/`.

219. Niklaus Wirth. “What can we do about the unnecessary diversity of notation for syntactic definitions?” In: *Communications of the ACM* 20.11 (1977), pp. 822–823.

220. Matthew Sowders and Jan Bækgaard Pedersen. “Mobile Process Resumption in Java without Bytecode Rewriting”. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Citeseer. 2011, p. 1.

221. Naftaly H Minsky. “Towards Alias-Free Pointers”. In: *European Conference on Object-Oriented Programming*. Springer. 1996, pp. 189–209.

222. *C++ Core Guidelines*. `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`.

223. Luca Cardelli. *Typeful Programming*. Digital Equipment Corporation Systems Research Center, 1989.

224. *Generics*.
http://www.eecs.qmul.ac.uk/~mmh/APD/bloch/generics.pdf.

225. Maurice Naftalin and Philip Wadler. *Java Generics and Collections: Speed Up the Java Development Process*. " O'Reilly Media, Inc.", 2006.

226. Kresten Krab Thorup and Mads Torgersen. "Unifying Genericity". In: *European Conference on Object-Oriented Programming*. Springer. 1999, pp. 186–204.

227. G Bracha, M Odersky, D Stoutamire, and P Wadler. "Adding Genericity to the Java Programming Language". In: *Proceedings of OOPSLA*. Vol. 98.

228. Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. "Towards an Existential Types Model for Java Wildcards". In: *Formal Techniques for Java-like Programs (FTfJP)* (2007).

229. Alexander J Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. "Towards a Semantic Model for Java Wildcards". In: *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*. 2010, pp. 1–7.

230. Ben Greenman, Fabian Muehlboeck, and Ross Tate. "Getting F-bounded Polymorphism Into Shape". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 89–99.

231. Edsger Wybe Dijkstra et al. *Notes on Structured Programming*. 1970.

232. William Shotts. *The Linux Command Line: A Complete Introduction*. No Starch Press, 2019.

233. Vadim Zaytsev. "The Grammar Hammer of 2012". In: *arXiv preprint arXiv:1212.4446* (2012).

234. Gordon A. Rose and Jim Welsh. "Formatted Programming Languages". In: *Software: Practice and Experience* 11.7 (1981), pp. 651–669.

235. Martin Bravenboer and Eelco Visser. "Guiding Visitors: Separating Navigation from Computation". In: (2001).

236. Tomasz Imielinski and Heikki Mannila. "A Database Perspective on Knowledge Discovery". In: *Communications of the ACM* 39.11 (1996), pp. 58–64.

237. Robin Cooper. "Records and Record Types in Semantic Theory". In: *Journal of Logic and Computation* 15.2 (2005), pp. 99–112.

238. John T Minor. "C Minor: A Pedagogical Language Based on High-Level Design Principles". In: *UNLV School of Computer Science Technical Report# CSR-18-001* (2018).

239. Melvin Fitting. "Tableau methods of proof for modal logics." In: *Notre Dame Journal of Formal Logic* 13.2 (1972), pp. 237–247.

240. Jeremy Avigad. "Raymond M. Smullyan, First-Order Logic". In: *Journal of Symbolic Logic* 61.1 (1996).

241. Patrick Borras, Dominique Clément, Th Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. "Centaur: The System". In: *ACM Sigplan Notices* 24.2 (1988), pp. 14–24.

242. Gilles Kahn, Bernard Lang, Bertrand Melese, and Elham Morcos. "Metal: A Formalism to Specify Formalisms". In: *Science of Computer Programming* 3.2 (1983), pp. 151–188.

243. Elham Morcos-Chounet and Alain Conchon. "PPML: A General Formalism to Specify PrettyPrinting." In: *IFIP Congress*. 1986, pp. 583–590.

244. Thierry Despeyroux. "Executable Specification of Static Semantics". In: *International Symposium on Semantics of Data Types*. Springer. 1984, pp. 215–233.

245. Thierry Despeyroux. "Typol: A Formalism to Implement Natural Semantics". PhD thesis. INRIA, 1988.

246. Gilles Kahn. "Natural Semantics". In: *Annual symposium on theoretical aspects of computer science*. Springer. 1987, pp. 22–39.

247. Gerhard Gentzen. "Investigations Into Logical Deduction". In: *American philosophical quarterly* 1.4 (1964), pp. 288–306.

248. Attali, Isabelle and Caromel, Denis and Oudshoorn, Michael. "A Formal Definition of the Dynamic Semantics of the Eiffel Language". In: (June 1998).

249. Pieter H Hartel. *LATOS a Lightweight Animation Tool for Operational Semantics*. 1997.

250. David Turner. "An Overview of Miranda". In: *ACM Sigplan Notices* 21.12 (1986), pp. 158–166.

251. Arie Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach*. Vol. 5. World Scientific, 1996.

252. Jan A Bergstra. *Algebraic Specification*. ACM, 1989.

253. Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. "The Syntax Definition Formalism SDF—Reference Manual". In: *ACM Sigplan Notices* 24.11 (1989), pp. 43–75.

254. Peter D Mosses. "Semantics of Programming Languages: Using ASF+ SDF". In: *Science of Computer Programming* 97 (2015), pp. 2–10.

255. Sandra Mara Guse Scós Venske and Martin A Musicante. "Typechecking XQuery: A Prototype in ASF+ SDF". In: *RECEN-Revista Ciências Exatas e Naturais* 8.2 (2006), pp. 247–258.

256. Benjamin C Pierce, Peter Sewell, Stephanie Weirich, and Steve Zdancewic. "It is time to mechanize programming language metatheory". In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2005, pp. 26–30.

257. Heinz-Dieter Ebbinghaus, Jörg Flum, Wolfgang Thomas, and Ann S Ferebee. *Mathematical logic*. Vol. 1910. Springer, 1994.

258. Thomas J Jech, Thomas Jech, Thomas J Jech, Great Britain Mathematician, Thomas J Jech, and Grande-Bretagne Mathématicien. *Set theory*. Vol. 14. Springer, 2003.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Benjamin Cisneros Merino
Email: benjcisneros@gmail.com

Degrees:

Masters of Science in Computer Science 2019
University of Nevada Las Vegas
Bachelor of Science in Computer Science 2017
University of Nevada Las Vegas

Thesis Title: Jiapi: A Type Checker Generator for Statically Typed Languages

Thesis Examination Committee:

Chairperson, Dr. Jan Bækgaard Pedersen, Ph.D.
Committee Member, Dr. Andreas Stefik, Ph.D.
Committee Member, Dr. Fatma Nasoz, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.